January 2016

TrackWiz: Visualizing Graphs of Tracks

Alec Benson
Worcester Polytechnic Institute

Patrick E. Lynch
Worcester Polytechnic Institute

VISUALIZING GRAPHS OF TRACKS

A Major Qualifying Project submitted to the faculty of the Worcester Polytechnic Institute in partial fulfilment of the requirements for the Degree of Bachelor of Science

Submitted by: Alec Benson, Patrick Lynch

Advised by: Professor Elke Rundensteiner

BAE Mentors: Keith A. Pray and Dr. Chris Smith

I. Abstract

BAE systems, a multi-national defense company, must interpret and visualize massive quantities of geo-temporal track data. While BAE currently has some tools that are capable of representing this data, existing tools cannot effectively represent association likelihoods between track entities. This project provides a software solution called TrackWiz for interacting with and visualizing this complex data and for interaction track associations.

Table of Contents

II. Acknowledgements

Without the help of our mentors and advisors, this project would never have been possible. Therefore, we would like to offer thanks to the many people that assisted us throughout the course of the project:

- Professor Elke Rundensteiner for all of the coaching, design insight, and advice throughout the project
- Keith A. Pray at BAE systems for offering continuous support and serving as a liaison between BAE Systems and WPI.
- Dr. Chris Smith for mentoring, and offering guidance and feedback throughout the project.
- BAE systems for providing us with the opportunity to work on this project.

III. Introduction

The motivation for this project was to develop an integrated graph and track visualization software for use with tracking and fusion engines such that both visualizations may be viewed concurrently and interacted with in a highly linked and integrated fashion. Graph visualization is an intricate field that involves displaying complicated data in a way that is understandable to humans [7]. Because of these challenges, we wanted the resulting software to be easily extendable, easy to use, and fast performing. The resulting software is known as TrackWiz, and runs in nearly all modern browsers.

TrackWiz was created in collaboration with the defense company BAE Systems. Throughout the course of this project, we received a wide range of feedback from BAE Systems employees that work with track and graph data on a daily basis. Through this feedback, we were able to define a concrete list of requirements. One requirement of our project is the ability to display the data in a variety of fashions such as on a spherical globe or on a rectangular map [2]. To accomplish this, we use the CesiumJS and D3 libraries. CesiumJS is a library used to display geographical data. The geographical data that is being displayed is geographical tracking data; geographical tracking data is a data representation of the position of an object which is being tracked. Specifically for this project, CesiumJS is used to display geographic tracking data. D3 is a data visualization library that has many great features for visualizing any data. In the context of this project, D3 is used to display an abstract graph.

Our team created software that not only dynamically visualizes sample simulation data, but also dynamically visualizes real world tracking data. The purpose of the CesiumJS library was to provide a real world context for the data while the D3 library is used to provide an abstract representation of the data [3]. BAE Systems uses tools to represent their geographic tracking data visually on a globe. In the background, they also have an abstract graph which, prior to this project, was never visually represented. This abstract graph is used to show how likely it is that two different geographic tracks are representing the same object. Displaying how correlated two different tracks are is useful to operators because different tracks may not entirely overlap with respect to time. If two tracks are representing the same object, and the two tracks are at different points in time, they may be combined in order to make a longer track which has the positional information of the object across all of the time that there is information about that object.

IV. Background

This section describes terminology that is utilized throughout the paper and it provides domain and background on tracking necessary for a full understanding of this project.

Sensors

The data used by this software originates from sensors. A sensor is any instrument that is capable of detecting information about its environment. Most sensors can be categorized into one of two categories: passive or active.

A passive sensor is any sensor that captures signals that are produced within the environment [1]. For example, a camera is a passive sensor because it collects light. It does not produce the signal that it collects.

On the other hand, an active sensor is any sensor which collects information about its environment by emitting energy. This energy is usually reflected back towards the sensor and collected [1]. Active sensors come in many forms -- radar, GPS, and sonar are all forms of active sensors. Typically, active sensors provide more information than passive sensors. However, they often require more energy to operate and are prone to interference when many active sensors are used at the same time [1].

Multi-Sensor Data Fusion

In most applications, a single point of sensor data does not provide enough information to be useful. However, by collecting hundreds, thousands, or even millions of data points, it is much easier to extract useful information about the environment. The utility of a sensor’s data is further compounded when it is combined with data collected by other sensors within the environment. The process of combining data from multiple sensors is known as multi-sensor data fusion.

Multi-sensor data fusion serves to improve the representation, certainty, accuracy, and completeness of the collected data [4]. Within the context of sensor fusion, representation describes the granularity of the data collected by the sensor. To be precise, fused data generally offers better representation than each individual data source does by itself.

Certainty describes the confidence whether the data obtained is accurate. For example, if multiple sensors all report the presence of a plane in the sky, then it can be said with greater confidence that a plane was actually there. *Accuracy* describes how close a sensor’s recorded data is to the actual value within the environment. Fusing data allows noisy or invalid data to be eliminated or disregarded. Completeness is a description of how much of the environment has been observed. By introducing new data from different sensors, the scope of the measured environment can be expanded. While each of these metrics are closely related and frequently interdependent, they are all equally important and distinct. In short, data fusion allows for much more information to be extracted from previously independent data sources.

Data Association

Before data fusion can be carried out, a relationship between data sources must be established. This process is called *association***,** and is a crucial part of interpreting sensor data [5]. To understand the importance of association, consider a self-driving car that determines how fast it should drive by measuring the speed of other vehicles on the road. If the self-driving car cannot distinguish a pedestrian from another vehicle, then it is likely to make incorrect and potentially dangerous assumptions when navigating. Similarly, a missile defense system must be able to discern enemy missiles from friendly ones. In the broadest terms, the goal of association is to probabilistically determine which measurements belong to any given target in an environment [5]. There are a variety of complex formulas that can be used determine the probability of association. While this MQP report will not cover them, it is still important to understand that data association is a crucial part of faithfully representing both *source* and *fuse tracks*.

TrackWiz Terminology

Throughout this MQP report, we use a variety of specialized terms. An accurate understanding of the meaning of each of these terms is necessary for a complete understanding of the remainder of the report.

First, we define an *operator* as any user of the TrackWiz software. More specifically, the term is commonly used to refer to either a programmer that is using TrackWiz to debug and verify the accuracy of data or a customer that is using TrackWiz to analyze geospatial data.

Within TrackWiz, the operator uploads data into logical groupings called *collections*. A collection may contain any number of data files. Collections are used in order to group data. Grouping the data allows operators to apply operations, such as toggling visibility, to the whole collection simultaneously.

A *visualizer* is software that graphically represents data. In the context of this project, the front end of TrackWiz is the visualizer. TrackWiz takes the uploaded data and represents it in a way that an operator may understand.

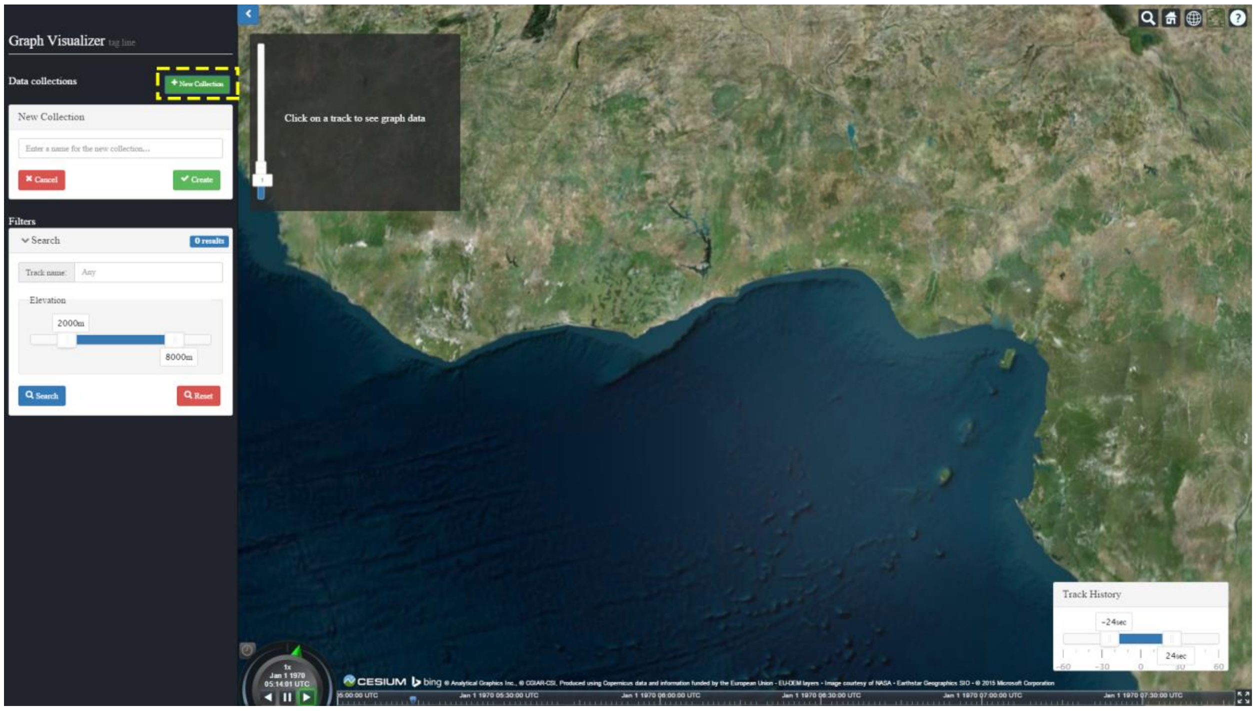
A *track* is what is being represented in TrackWiz. A track is the data that is collected when an object is being tracked. This data has two aspects. The first is geotemporal in nature; it shows where the object was at a given time. The second aspect represents how likely it is that two separate tracks are representing the same object.

A general definition of a *graph* is a collection of vertices and edges. Within the context of TrackWiz, a graph refers to an abstract representation of the tracks. This abstract information allows operators to figure out if two different tracks are related. Each node in a graph in TrackWiz represents a track. An edge connecting two nodes shows that the two tracks are related in some way. There is a weight associated with each edge; this weight shows how related the two tracks are.

V. Major Design Goals

In order to create intelligently designed software, it is crucial that interactions with the software are well defined. Requirements for TrackWiz were gathered through a multi-step process. First, we examined some of the software that is currently in use by BAE systems. Secondly, we interviewed our mentors and identified further difficulties with the existing software. Using this knowledge, we compiled a preliminary list of goals and prioritized the list by necessity and difficulty of implementation. Finally, we extracted the major features from the list. These goals are defined below.

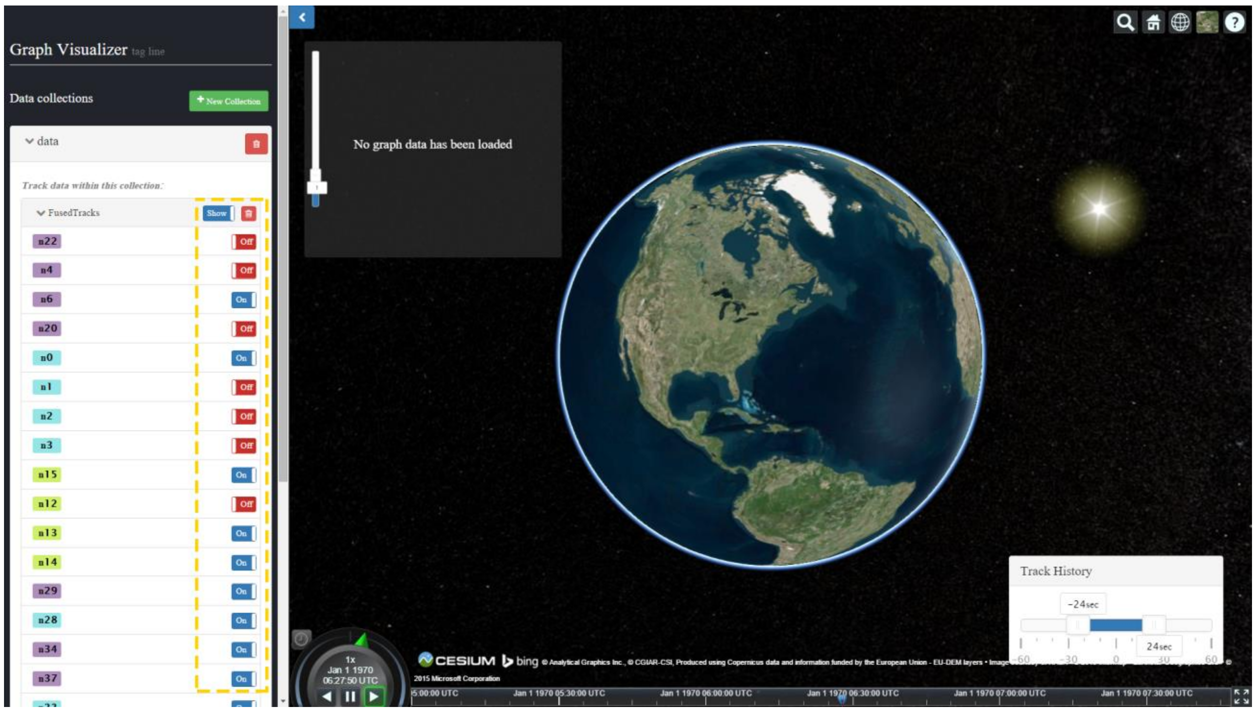
Managing data collections in TrackWiz



While analyzing the existing software used by BAE Systems to manage track data, it became clear that the software did not have an effective means of grouping and organizing data sets. This is an especially important task because similar data may be spread across multiple files. Further, because the user may be visualizing tracks from tens or even hundreds of files, it is critical that users be able to logically group data with similar characteristics.

This goal was achieved in TrackWiz through the use of the *collection* system. As the name suggests, a collection represents a group of track data. The collection interface not only allows grouped data to be visually toggled, but also enables users to upload custom 3D track models to represent all track entities within a collection. To add a new data source, the user will first click on the “New Collection” button within the collection panel. After doing so, the user may select the type of data to upload to the server (either graph of track data). In the event that a track file is provided, the user simply clicks a “browse” button on the collection panel, locates the XML data on his or her computer, and then clicks “upload”. In the event that a graph file is chosen, the user follows the same process but for the Sage graph file.

Expanding and collapsing of data collections



No graph data has been loaded

Data sources may consist of tens of thousands of data points. With this in mind, we decided that users should have the ability to collapse large sets of data into smaller, more manageable, and more visually understandable chunks. When an expanded node is clicked on in the viewer, the node and its children are combined into a single node. Collapsed nodes should be visually represented differently than non-collapsed nodes. First of all, the size of the node is dependent on the number of children that it contains. Put simply, a node with 100 children nodes should be larger than a node with only a dozen children nodes.

When a collapsed node is clicked on in the viewer, the node expands to show itself and all of its children nodes. The children are connected to the parent node via edge lines. The thickness of the edge line is directly proportional to the strength of the association between the two nodes. Nodes with stronger associations are drawn with thicker lines, and nodes with weaker associations are drawn with thinner lines.

Data sources should be representable via both D3JS and CesiumJS

Both D3.js and Cesium.js are powerful tools. However, each of the two libraries possesses unique strengths and weaknesses. For instance, Cesium JS is excellent at presenting data within three geographic dimensions. However, it is not able to display graphical representations outside of this niche. On the other hand, D3.js is not effective at drawing geographic data in three dimensions but it is far more effective at representing two dimensional charts and graphs. In this way, each library makes up for the shortcomings of the other. Because of this, our application takes advantage of both tools. Cesium JS is used to draw the geographic data, and D3.js is used to chart and visualize the statistical characteristics of the data sources. More specifically, D3.js can be used to provide information about the strength of an association of data within the graph, the changes in speed and elevation of a data set over time, and the total distance of the track.

Preventing information overload

The quantity of data that must be visually representable by the software introduces a major concern: “how can many thousands of points of data can be shown to the user without overly cluttering the screen or overwhelming the user?” With this goal in mind, many design decisions needed to be made to represent the data in complete and meaningful ways without introducing comprehension overload.

This goal was kept in mind when designing all of the components of TrackWiz. Our design of the abstract graph is one of the most prominent examples of our efforts to limit the amount of data that users must interact with to use TrackWiz. The graph slider reduces the amount of information that operators must look through and further enhances usability because it only shows associations for data that users are currently interacting with.

To this end, the search controls also allow operators to locate specific points of interest within the data without having to manually sort through all of the on-screen data. For instance, if an operator is specifically looking for tracks that have an elevation between a certain range, the search filter allows the operator to find and display tracks that meet this requirement. Further, the operator may locate specific tracks by entering either the full name or a substring of the name of a track in the search bar. All tracks that contain the provided string in their name are highlighted in the Cesium viewer.

Our efforts to reduce information overload are also evident in our design of the data collection control panel. In this panel, users may separate uploaded data into as few or as many collections as desired. This level of control makes it much easier for operators to arrange data in ways that make sense for the task at hand. Additionally, each collection panel can be expanded or collapsed. This feature makes it easy for operators to focus on data within a specific collection without having to keep irrelevant or uninteresting data sets in the view. Similarly, each data file within a collection may be expanded or collapsed as well. This feature is particularly vital because files may contain hundreds or even thousands of different track elements.

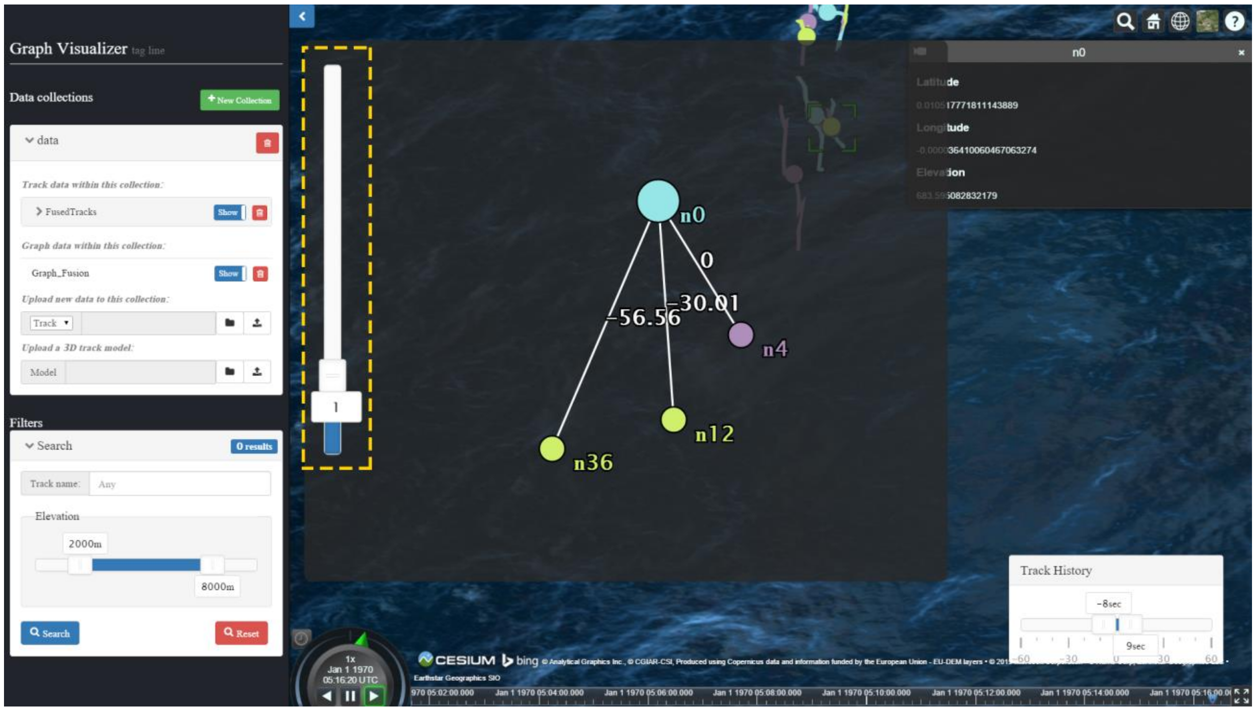
Within the expanded view of a track file, operators may toggle the visibility of each of the individual tracks within a file. Additionally, if a user wishes to toggle the visibility of all tracks within a file at once, the user can simply click on a toggle switch next to the name of the file. This level of control also exists for uploaded graph data. If a user has uploaded a graph file that is no longer of interest, its visibility may be turned off by clicking on the toggle box within the collection view pane. The ability to toggle the visibility of tracks and graph files and each of the sub elements makes it easy for operators to focus on specific data. Giving users the ability to toggle the visibility of data allows them to work exclusively with data that is of interest.

Finally, the track history control panel also allows users to control the amount of information that is displayed on the screen. This control panel contains a single slider with two handles. The units of the slider represent time in seconds, and dragging the handles increases the length of the trail that is displayed in front of or behind tracks within the visualizer. For instance, setting the leftmost handle to “-60 seconds” and the rightmost handle to “+60s” will show a line coming out of track nodes that represents the location of the tracked entity up to a minute in the future and a minute in the past. Giving users the ability to adjust the length of these track lines is useful as there may be many tracked objects in close proximity. In this case, the user may want to adjust the length of the lines to reduce screen clutter and make the data easier to read. On the other hand, if a user is interested in studying the motion of a track over time, the user may want to see more track history. The user can do so by adjusting the track history slider as desired. We found that the ability to control the length of these track lines added a great deal of usability to the software. Also, it made complex data much easier to understand and interpret.

VI. Major Design Principles of the TrackWiz Software

In this chapter, we describe in detail the important considerations that were made when creating TrackWiz.

Abstract Graph Representation



One of the most important aspects of this project is to decide on a method to display abstract graph information to users. This task was especially challenging because BAE systems did not have a tool for rendering a graph that we could evaluate and draw inspiration from.

Nonetheless, we had a strong understanding of the data that would be loaded into the graph. This understanding was acquired by talking with the BAE system mentors and closely examining the existing software. This knowledge was used to guide our design decisions.

First and foremost, we recognized that the data could potentially consist of hundreds or even thousands of vertices and edges. Therefore, there must exist an easy interface for controlling the amount of data that is shown in the graph at any given time. To solve this problem, we decided that the graph should only display graph information about data that the user is interacting with. In our solution, the graph is initially empty until the user selects track data within the Cesium Viewer. When this happens, the graph displays the selected track as a colored circle.

Next, we needed to describe the design process for displaying associations between selected tracks. Since the node could be associated with an unspecified number of other nodes, we wanted to provide the user explicit control over the quantity of associations to display. We solved this issue by placing a slider on the side of the graph display. The slider values range from zero to five, where one is the default value. The user may adjust the slider values by clicking on the slider handle and dragging it either upwards or downwards. The slider controls the association depth to display. For instance, at depth one, the abstract graph will display the selected node and all associated nodes that are directly adjacent to it. At depth two, the slider shows an additional layer of nodes and edges stemming from the immediately adjacent nodes of the selected node. Therefore, when the graph is all the way at the bottom at value zero, the operator just sees the selected node on the abstract graph. When the slider is moved all the way to the top at value “1”, the operator sees the selected node and all nodes that are at most five degrees of separation from the selected node. When representing real data, the abstract graph is very large and will be a dense graph. Due to these properties, displaying any more than five degrees of separation would be incomprehensible. Using this solution, we were able to greatly reduce the number of nodes and edges that the user must see when interacting with the abstract graph.

Despite these efforts to limit the number of entities that a user must examine within the abstract graph, we still recognized the possibility that an exceedingly large dataset that contains tracks with many adjacent nodes could still potentially overwhelm the user. To remedy this, we designed the abstract graph so that users could easily zoom and pan through nodes and edges. Therefore, if the graph was especially large, the user could simply zoom out to see a larger portion of the graph, or alternatively zoom in to focus on specific details of the graph. If the user is zoomed in and would like to look at another portion of the graph, the user may also click and drag on any blank space within the graph to pan in a direction.

We also put much thought into the way that nodes are represented within the graph. The first design decision was how to make any given node distinctly recognizable from another node. In our early prototypes, we assigned a unique color to each node in the graph. However we determined then that this provided little usefulness to operators and easily overwhelmed them. We quickly revised the system to color nodes based on the sensor that collected the track information. This color is consistent across all displays and further helps users locate and identify tracks across the different representations. Further, since there is no theoretical limit on the number of nodes that the user may upload, we need to establish a unique system for generating colors. Our system is to make our software generate the node color by hashing the track's "sensor type" and "sensor platform" fields into a deterministic hex color. This means that our system is able to assign a unique color to each track -- barring occasional hashing collisions.

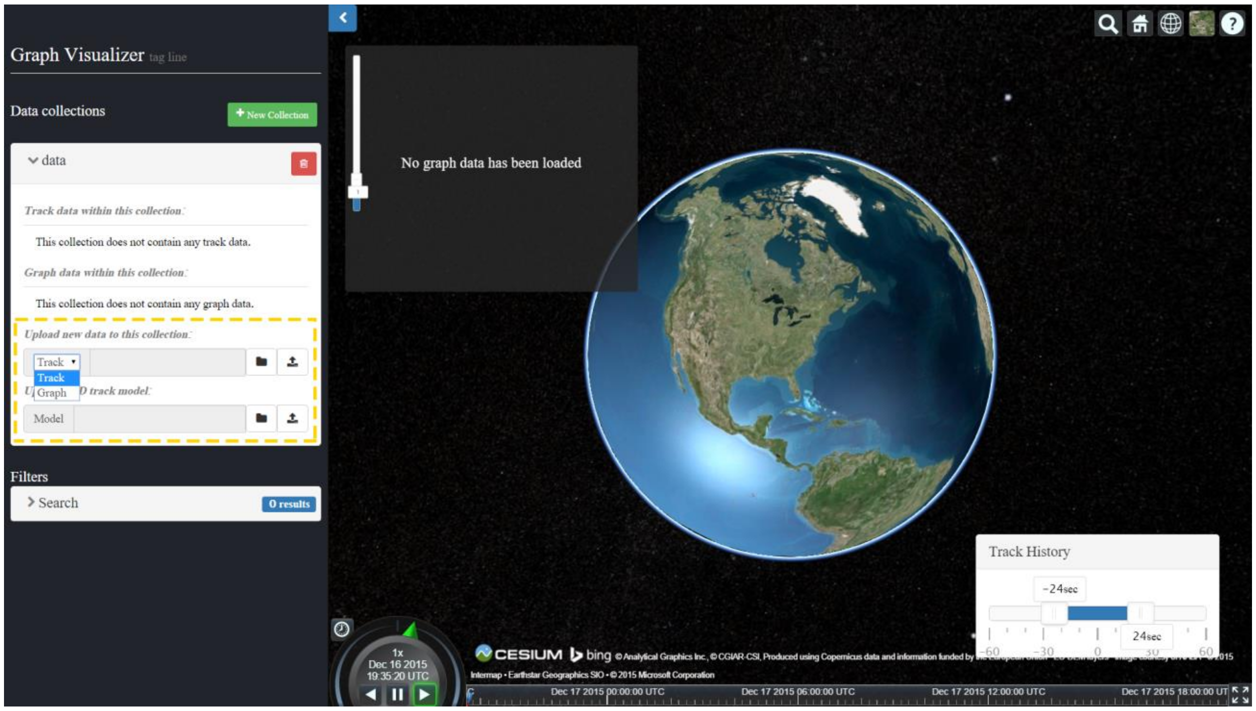
However, we did not want users to have to rely solely on the color of the track to identify it. Therefore, we also labeled each node in the graph with its track label. To further enhance readability, the node labels are given the same color as the node that they represent.Further. they are given a thin black outline to make them easily distinguishable if the generated color happens to blend with the background.

To facilitate the user to easily locate the node in the graph that represents the currently selected track, we decided to give this node the largest diameter of all nodes within the graph. In this way, the selected node is easily identifiable. This makes it clearer to users where the graph will expand when the graph slider is adjusted.

The design of the edges was also seriously considered for displaying the abstract graph data. Research and experimentation was done with displaying the edges as well as the edge weights. In the initial design, the edges varied in width with edge weight and edge weights were not displayed. As the project evolved, it became clear that it was necessary to display the edge weights. The graph became cluttered when the edge weights were displayed, so the edges were changed to vary in length rather than in width. This cleared up the clutter problem but another problem was realized when the clutter was cleared. Namely, it was sometimes difficult to read the edge weights, which were just white text at that time. The edge weight hovered over an edge, with all edges being white. This problem was more complicated than just changing the text color because the background of the graph is a very dark color. Thus there is not a single color that would be easy to read. The solution to this problem was to keep the text white, but give it a black outline. This would allow the text to be easily read regardless of what appeared behind the text.

Finally, the interactions with the abstract graph would synergize with the existing interfaces. To do so, in our design when one clicks on any node within the abstract graph, this centers the Cesium viewer on this track. We found that this design scheme worked very well. It allowed operators to examine large sets of data within a short period of time.

File Upload


Graph Visualizer tag line
Data collections
New Collection
data
Track data within this collection:
This collection does not contain any track data.
Graph data within this collection:
This collection does not contain any graph data.
Upload new data to this collection:
Track
Track
Graph
D track model:
Model
Filters
Search
0 results
No graph data has been loaded
Track History
-24sec
24sec
CESIUM
bing
Dec 16 2015 19:35:20 UTC
Dec 17 2015 00:00:00 UTC
Dec 17 2015 06:00:00 UTC
Dec 17 2015 12:00:00 UTC
Dec 17 2015 18:00:00 UTC

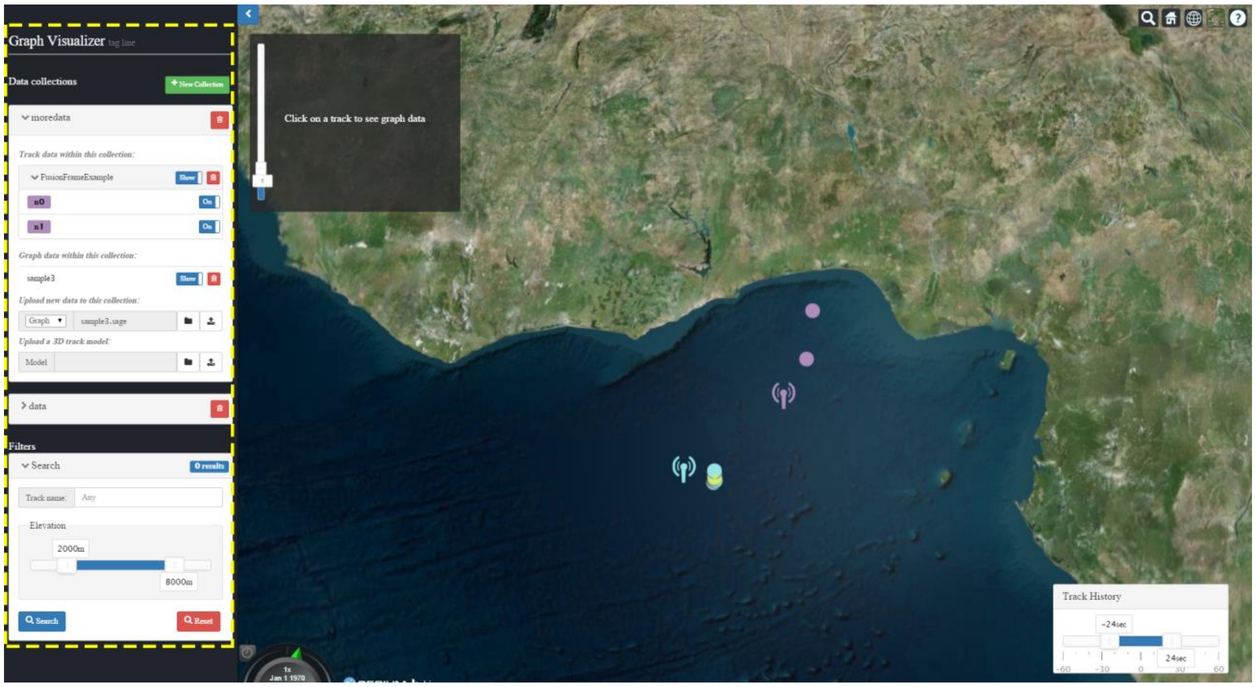
The interaction with the file upload controls were designed such that a user only needs to interact with a single area in order to improve the user experience. The location of the file upload is at the bottom of a collection. In this area, a user may upload track or graph data as well as a model to represent the collection. To upload track data, the user selects “Track” from the drop down box, clicks on the folder button to open a file explorer, selects a file, and clicks the upload button. To upload graph data, the user follows almost the same process. Rather than selecting “Track” the user selects “Graph” in order to upload graph data. For a user to upload a model to a collection, the user will click on the folder button, select a model file, and then click the upload button.

Security was carefully considered while designing the file upload because file uploads are a common attack vector for malicious users. This is because a file upload system allows a user to put arbitrary, potentially malicious, files onto the web server. To handle file uploads on the server side, multer was used. Multer was tested for security vulnerabilities such as file traversal, both absolute and relative. In order to prevent users from overwhelming the application with exceedingly large files, there is a maximum file size limit of 100MB per file. If the file is too large, the user is informed via an error message that appears below the upload controls. Data may be provided to TrackWiz in any one of three formats: XML, Sage, or GLTF. We also wanted to ensure that TrackWiz would be safe from external entity attacks. In an external entity attack, the malicious user attempts to include another file in the XML, such as /etc/shadow, to make the information from that external file available in the XML, so that, when parsed, the information becomes available to the malicious user.

VII. Evaluation

This section describes the design and implementation decisions that were made throughout the course of creating the visualization software to represent the track data.

Colors in the User Interface



At the start of this project, there was not a consistent color scheme. This is because we used controls from many packages used each of the packages had different default colors. For example, the default colors of the toggle button that was used to hide or display different tracks are blue for on and gray for off. However, the default colors for the create and cancel buttons used in collection creation are green and red, respectively. After receiving feedback, we decided that there should be a consistent color scheme. Each color was given a meaning and the controls were modified so that the colors of the controls fit that meaning. Green means “confirmation” or “success”. Blue means “information”. Finally, red means “error” or “negative”. Consistency of color is useful because it allows users to learn how to use the controls more quickly. It allows the user to learn how to use one control, then apply the same reasoning to all of the controls that are the same color.

The use of a consistent color scheme is evident throughout the UI. For example, the box that displays the number of search results is blue, indicating that it provides information to the user. Similarly, the slider that controls the level of association to show within the graph is blue, indicating that moving the slider adjusts the amount of information that the user interacts with. Because a consistent set of colors has been implemented, a user may now learn how to use the various controls of the project with more ease.

Performance Benchmarks

In this section, we analyze the time it takes to load a track file into TrackWiz. To measure the performance of the track upload process, we picked three track files of varying sizes. We then measured the total time it took to parse and display the track data. Each file was tested five times to ensure the reliability of the results.

We found that our smallest track file, which was 5kb in size, took an average of 0.72 seconds to display. The largest file, which was nearly 12 megabytes in size, took an average of 5.6 seconds. These results show that loading time is not linearly proportional to the file size. This is likely because the track loading process incurs a constant overhead that is not related to file size. As file size increases, this constant overhead becomes more negligible.

Track File Size	5kB	847kB	11,775kB

1	0.76	0.99	5.69
2	0.75	0.83	5.41
3	0.74	0.89	5.77
4	0.69	0.94	5.74
5	0.67	0.9	5.43
Average	0.72	0.91	5.608
Average data rate	6.9 kB/s	930 kB/s	2100 kB/s

This is good news: as the file size increases, so does the average data rate. This means that operators should be able to upload large track files with satisfactory performance.

While we did not have any larger track files to test our application with, we are happy with the performance of the upload process, and believe that TrackWiz should be able to handle files of even larger size with similar results.

VIII. Future Work

This project is not a small undertaking. We acknowledge that there are features that we were not able to implement within the given time frame. This section details some of the additional features that are worth implementing in the future.

Dynamic Abstract Graph

An aspect of the data that was not captured in the project is that the abstract graph changes with time. Implementing this would not be difficult as the abstract graph code, as well as the rest of the project, was written in a very extendable fashion. The parser would have to be written to accommodate the temporal information.

Collaborative Usage

One advantage to web applications is that many users may simultaneously connect to the application. Fully realizing the advantages of this was out of the scope of this project. There are features implemented utilizing the facets of a web application, but there is more that can be done.

Currently, an operator may see the data that other operators working on the same web server have uploaded. This has its advantages as well as its disadvantages. An advantage of this feature of a web application is that operators may collaborate with each other about their work. Unfortunately, if you imagine two operators each with different levels of security clearance, one operator may be analyzing data that the other operator would not be cleared to view. Though this may sound problematic, one potential, though non-ideal, solution is to have more than web server running at a single time. This way, the data would be segregated and the operator with sensitive data would be able to set their web server to only run locally, that is the server would only accept connections from the local machine.

A more ideal solution, though requiring significant effort, would be to implement authentication and authorization with regards to the data that is viewable by an operator. This would be ideal because then multiple operators may connect to the same server and only view the data that the operator is authorized to view.

The difficulty in adding the features of authentication and authorization would be both in the design and its implementation. The design would be difficult, because, currently, TrackWiz has no concept of permissions. Permissions would be a security requirement of the application because an operator would want to be able to share data with certain operators but not others. Simply, if an operator had permission to view a set of data, that operator would be able to view it but if an operator did not have permission, the operator would not be able to view it. There would need to be an arbitrary number of levels of authorization. This is necessary so that an operator may be able to pick and choose which other operators have access to which of their data sets.

Streaming Data

Currently, an operator is only able to upload a file of a complete data set which was collected prior to the operator viewing it. A feature that is out of the scope of the time frame for the project is the ability to point the web application towards a stream of incoming data that is rendered as the data is received. This would allow for the possibility of operators analyzing data in real time. Imagine a mission is happening and there are sensors collecting data on soldiers’ positions and enemies' positions. Being able to remotely view the situation in real time would allow operators to analyze the data as the mission is occurring which would lead to more informed decisions.

The difficulty regarding this feature lies in the implementation details.When streaming data, there are some formats that are better than others. For example, the tracking data is formatted in such a way that there would be very little overhead if the data were to be transmitted in real time. This is because the highest level of grouping in the data is by the time in which the data taken; this would allow a single time slice to be sent at a time. Inversely, the graph data is not formatted in an ideal way. It is formatted where the top level of information specifies whether the following is either a set of vertices or edges. This means that, if the data were being streamed in real time, there is not a good way to chunk the data as it is being taken in order to be sent to the web server.

IX. User Manual

In this section, we offer detailed instructions on using TrackWiz. This manual is intended for users that would like to learn how to fully take advantage of the software.

Installing and Running

Installing and running this project follows 4 steps:

1. Install Node.js and NPM
2. Download the source code
3. Install the Node dependencies
4. Run the project

Install Node.js and NPM

If you are one Windows or a Mac, download Node.js from https://nodejs.org/en/ and any version that is 4.2.2 or later should be sufficient. Run the executable in order to install both Node.js and NPM.

If you are running a Linux distribution, use your choice of package manager to install Node.js and NPM. Examples can be found at https://nodejs.org/en/download/package-manager/ for almost every package manager. Note that with certain package managers, such as apt-get, Node.js and NPM are bundled together, but on other package managers, such as pacman, Node.js and NPM are in separate packages.

Cloning the Repository

At the time of this writing, the source code is located at https://github.com/alecbenson/BAE-MQP so any commands that use that URL may need to be replaced once the project has found a permanent home. There are multiple ways to download the source code. This manual will cover cloning the repository using Git and downloading the source code from Github's web page.

Cloning the repository with Git requires Git to be installed. This manual assumes git has been added to your PATH environment variable. Navigate to the directory that you wish for the repository to reside. Run the command "git clone https://github.com/alecbenson/BAE-MQP " and the source code will be downloaded. Once the code is downloaded, there will be a new directory name "BAE-MQP" which contains all of the source code.

Downloading the source code through Github's web page is also quite simple. Use a web browser to navigate to https://github.com/alecbenson/BAE-MQP and click on the button labeled "Download ZIP" which will download a .zip file and put it in your downloads folder. Prior to moving on to the next step, move the .zip file to the directory in which you want the source code to reside, and unzip it.

Installing the Node Dependencies

Beginning with this step, and continuing until the Node.js web server is running, these instruction will be for a command line. The instructions will not be dependent on which command line is being used but it will be assumed that node and npm are on the PATH environment variable.

Navigate into the folder containing the source code that you cloned or downloaded and unzipped in the previous step. Next, navigate into the app folder which contains the source code for the web application. Execute the command “npm install .” and note the period at the end of the command. This will download and install the packages that this project uses.

Running the Project

The final step is to start running the project. When the project is run, a web server will be started. In order to start the server, run the command “node app.js” while in the app directory. The app directory is the directory which was navigated to in the previous step, immediately prior to running the “npm” command. This will start a local instance of the web server. You may connect to TrackWiz by opening a supported web browser and navigating to *localhost:8080*.

Users may also want to run TrackWiz publicly so that users outside the local network may connect to the application and upload data. To do so, simply use the --*public* flag. Typing “node app.js --public” will start a public instance of the server. Users may connect to TrackWiz by opening a browser and typing in the IP address or domain name that the server is being run on.

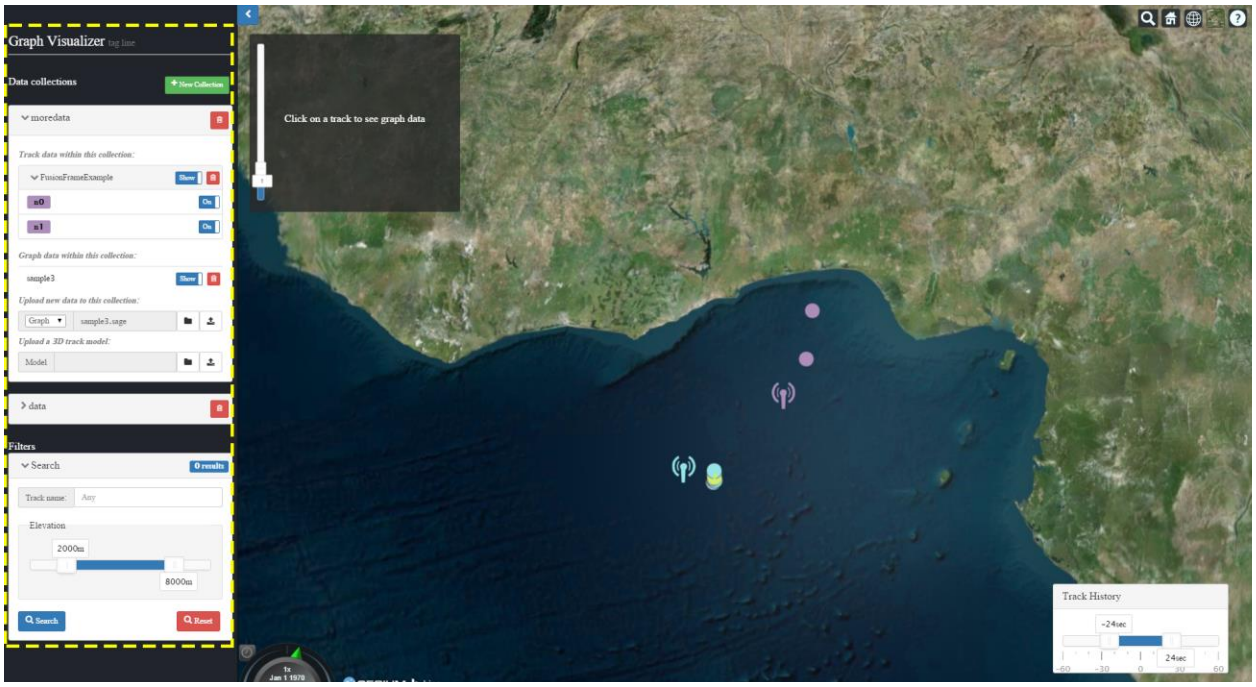
Using the Web Interface

This section discusses all of the features available to a user of this project. These features are:

1. Controls Sidebar
2. Making a new collection
3. Uploading files
4. Interacting with uploaded data
5. Track replay controls
6. Search

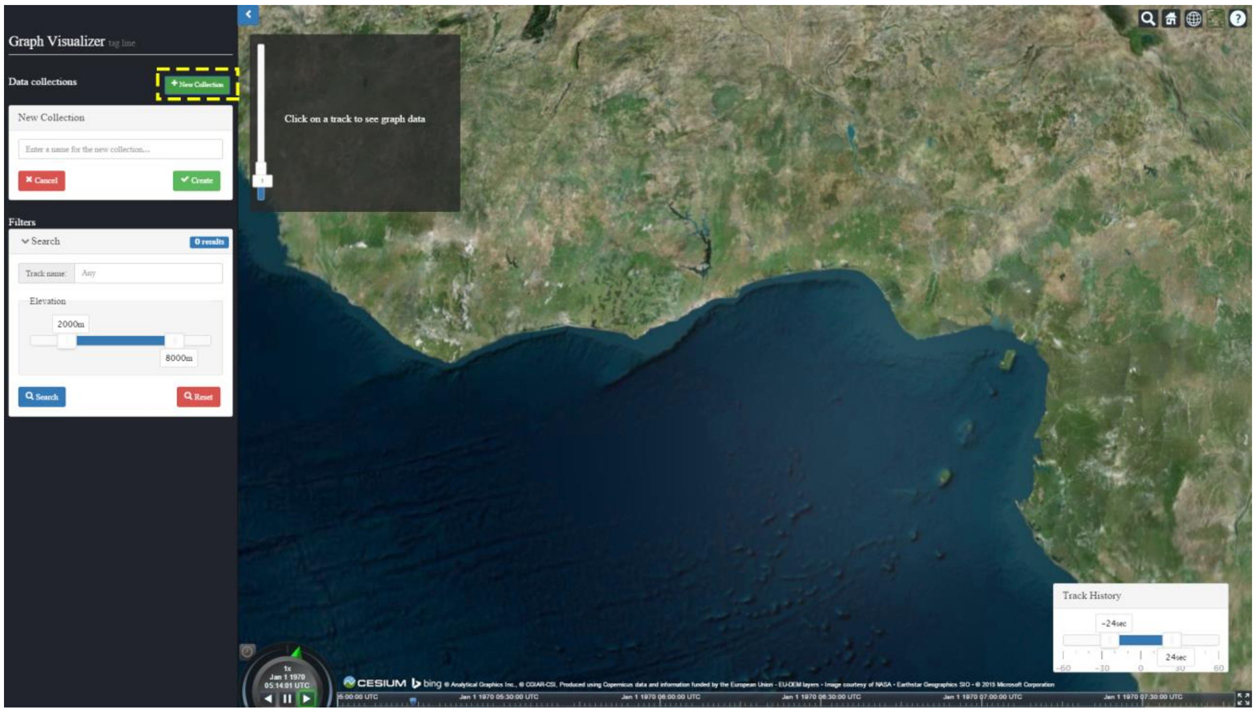
Controls Sidebar

Throughout this manual, a “controls sidebar” will be referenced. This is a panel on the left side of the screen which contains the majority of the controls used to interact with this project. In the image below, the yellow outlined section is the controls sidebar.

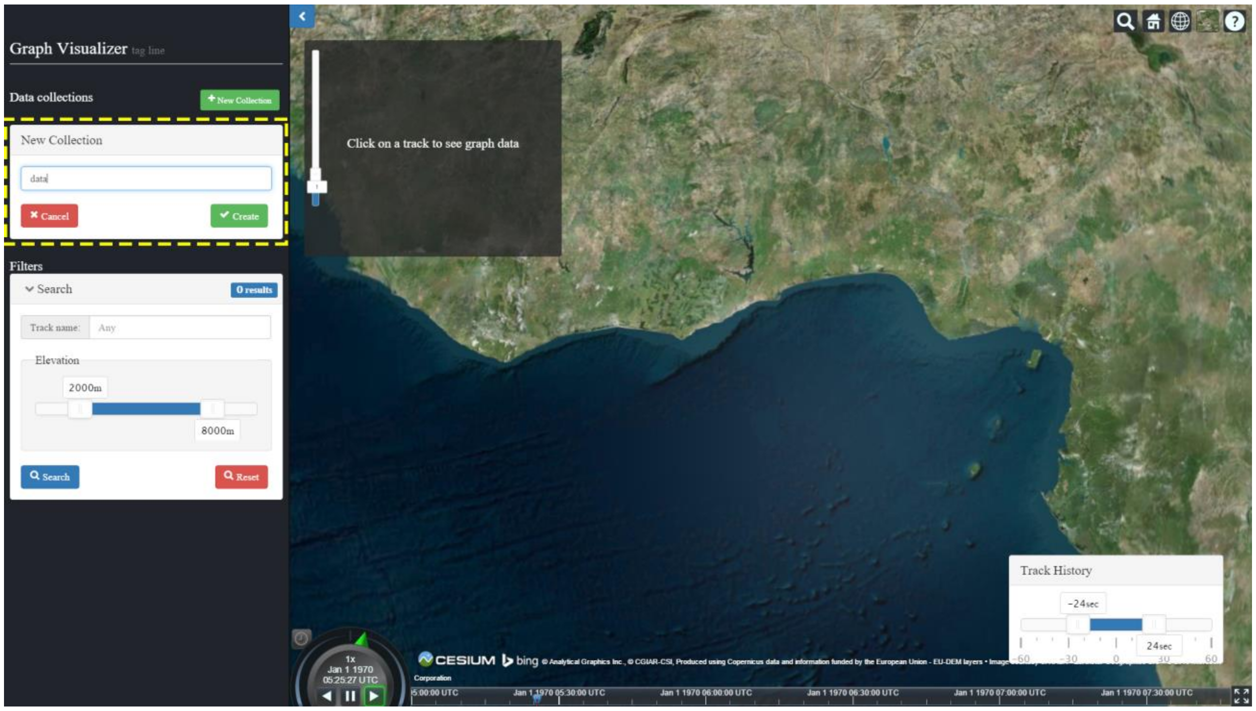


Making a New Collection

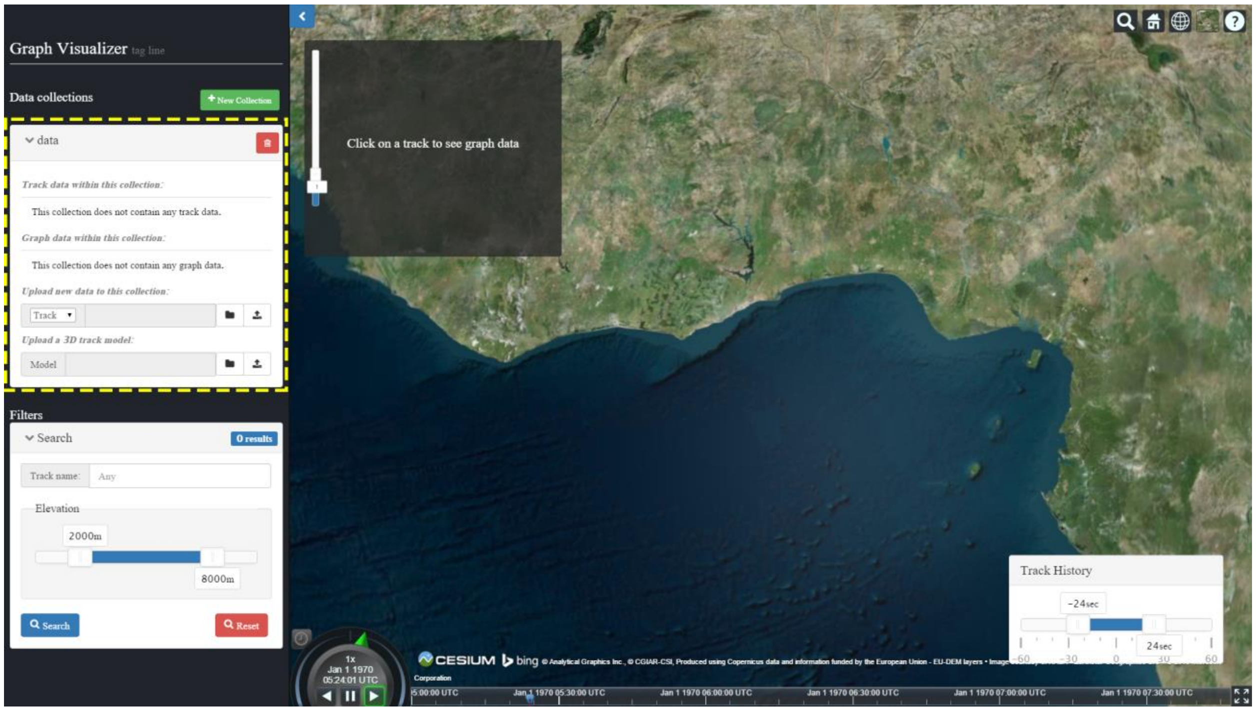
A collection is a structure which is used to group together sets of data so that a user may manipulate properties of the entire set at a single time. In order to make a new collection, a user clicks the green button labeled “New Collection” towards the top of the controls sidebar.



The project prompts the user to input a name for the collection.



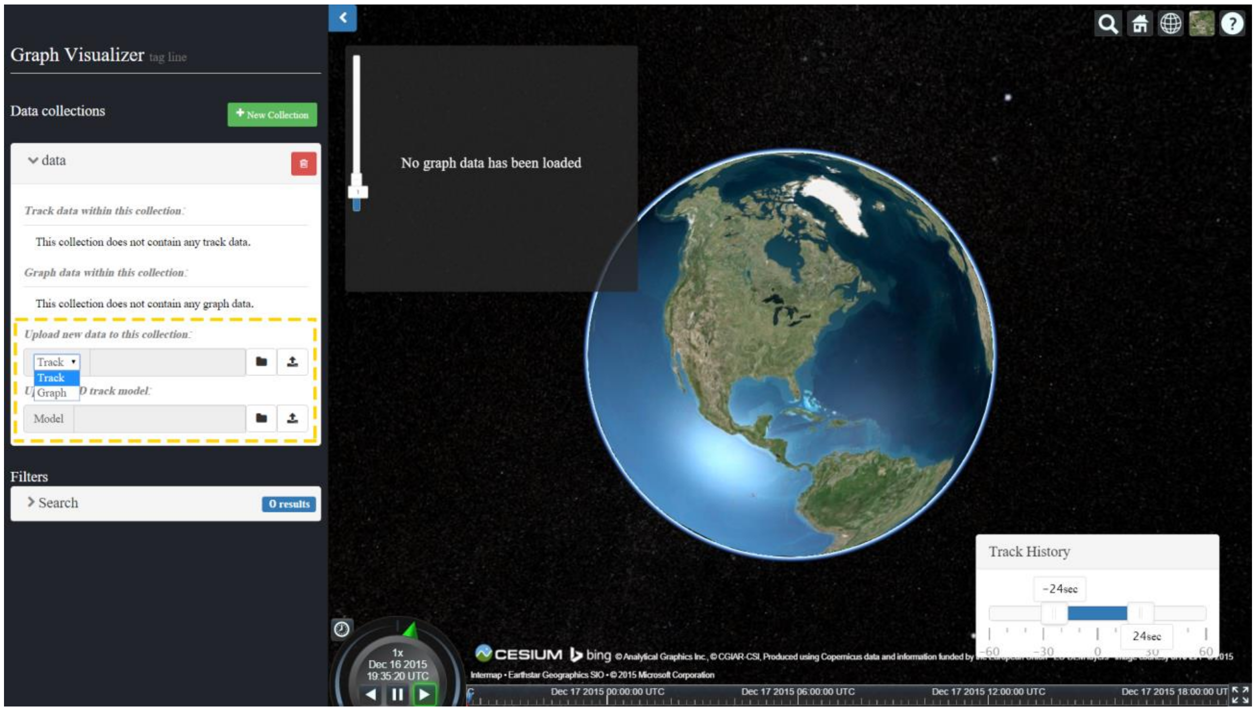
At this point the user may click the red cancel button to cancel creating the new collection or input a name and click the green create button to make a new collection appear. Once the collection is created, a box formatted with the track data, graph data, and a file upload section is created in the controls sidebar.


Graph Visualizer tag line
Data collections
New Collection
data
Track data within this collection:
This collection does not contain any track data.
Graph data within this collection:
This collection does not contain any graph data.
Upload new data to this collection:
Track
Upload a 3D track model:
Model
Filters
Search
0 results
Track name:
Any
Elevation
2000m
8000m
Search
Reset
Click on a track to see graph data
Track History
-24sec
24sec
1x
Jan 1 1970
05:24:01 UTC
CESIUM
bing

Uploading Files

Once a collection is created, a user needs to populate the collection with data. To do so, the user uses the controls that are beneath “Upload new data to this collection” within the collection. In this manual, uploading data is covered first then uploading a model is covered.

To upload new, geographic track data, a user interacts with the upload controls that have a drop down box next to them. This drop down box either says “Track” or “Graph” and that label corresponds with the type of data the user wishes to upload.

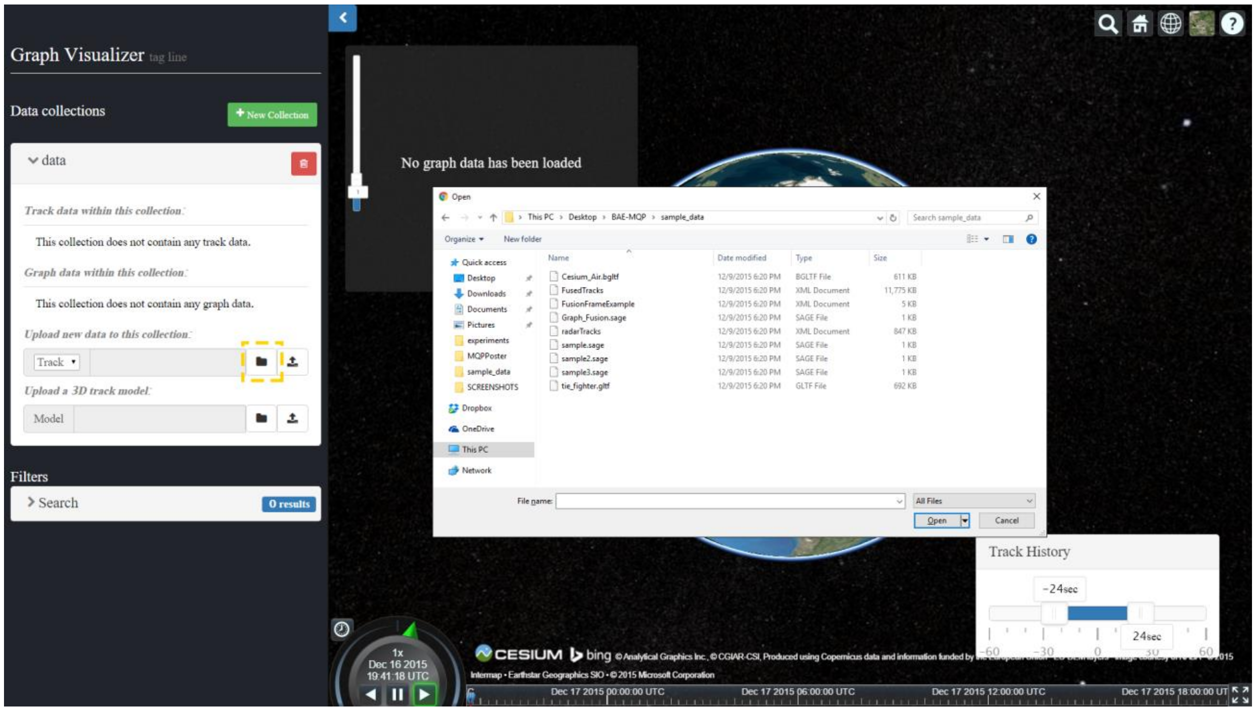


This collection does not contain any track data.

This collection does not contain any graph data.

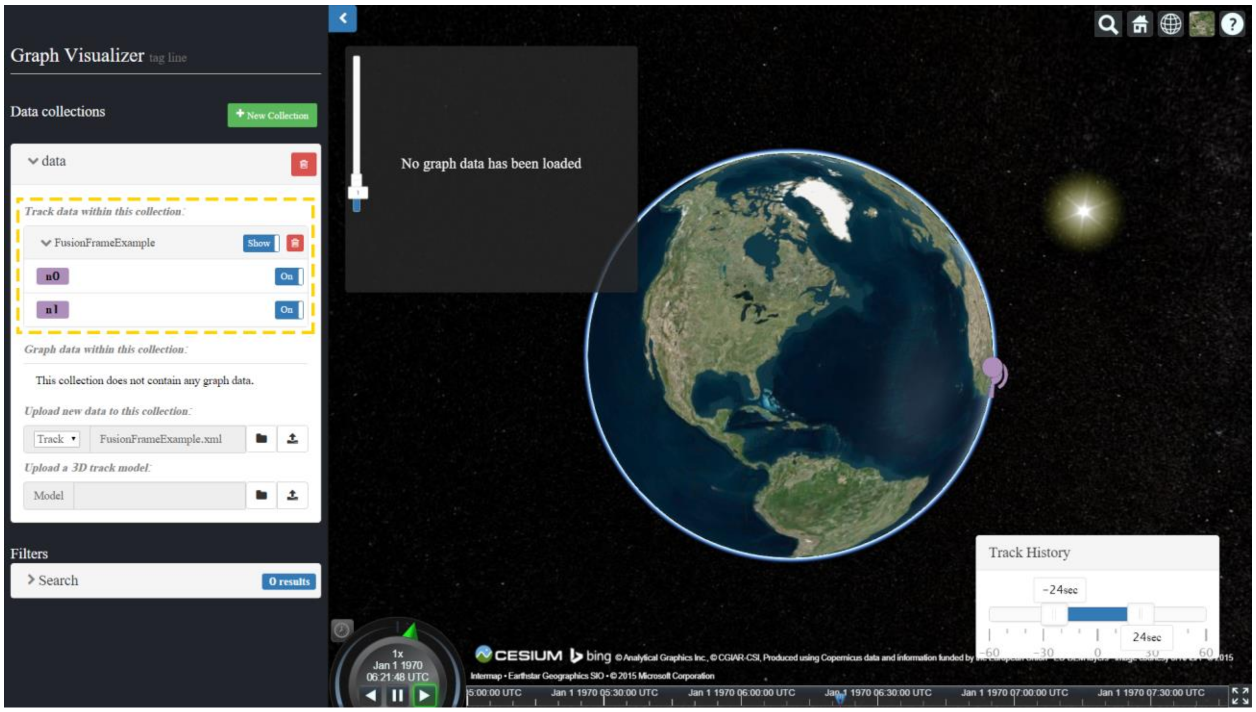
No graph data has been loaded

The track data that the user uploads must be an XML file. This data represents the geographic information related to the tracks. To select a file to upload, the user clicks on the button that is labeled with a folder icon. This is bring up the user’s file explorer from which the user will select a file.



Name	Date modified	Type	Size
Cesium_Air.bgltf	12/9/2015 6:20 PM	BGLTF File	611 KB
FusedTracks	12/9/2015 6:20 PM	XML Document	11,775 KB
FusionFrameExample	12/9/2015 6:20 PM	XML Document	5 KB
Graph_Fusion.sage	12/9/2015 6:20 PM	SAGE File	1 KB
radarTracks	12/9/2015 6:20 PM	XML Document	847 KB
sample.sage	12/9/2015 6:20 PM	SAGE File	1 KB
sample2.sage	12/9/2015 6:20 PM	SAGE File	1 KB
sample3.sage	12/9/2015 6:20 PM	SAGE File	1 KB
tie_fighter.gltf	12/9/2015 6:20 PM	GLTF File	692 KB

Once a file has been selected, the file’s name will appear in the upload box. To upload the data, the user clicks on the button that is labeled with a computer that has an arrow pointing up out of it. While the data is being uploaded, a loading overlay will appear momentarily. Once the overlay disappears, the track data will appear on the globe and the filename without the file extension will show up in the “Track data” section of the collection.

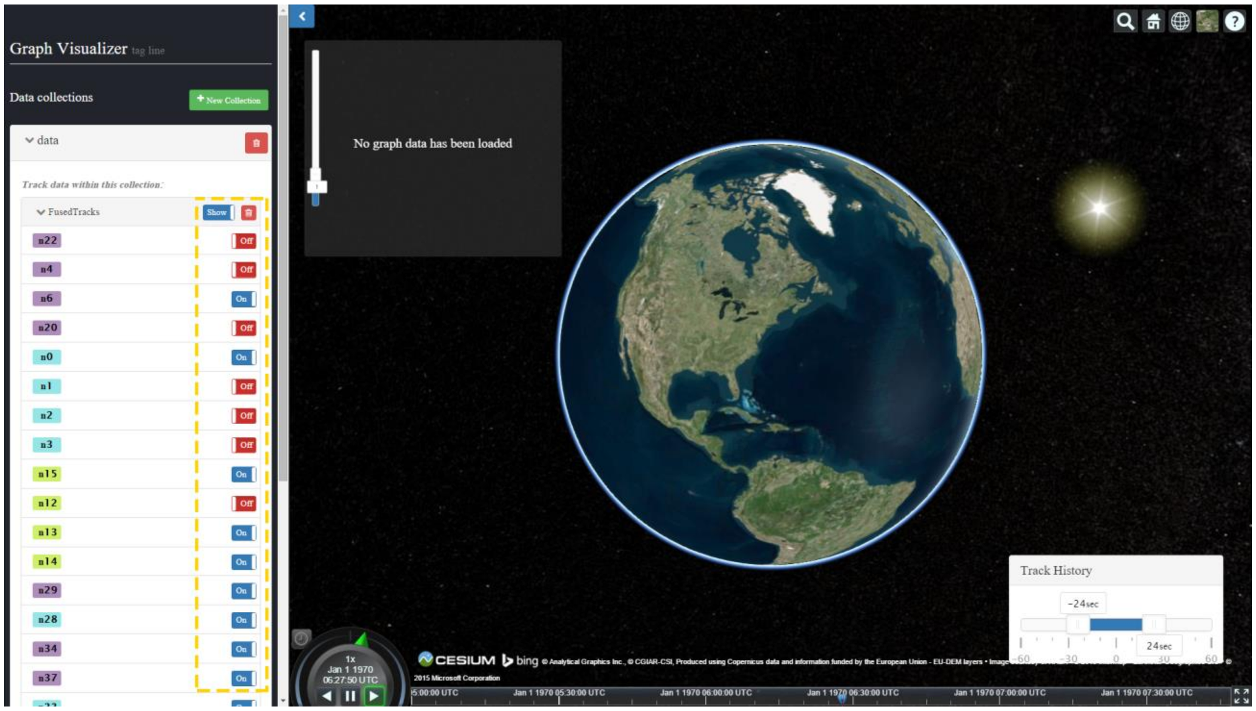

Graph Visualizer tag line
Data collections
New Collection
data
Track data within this collection:
FusionFrameExample
Show
n0
On
n1
On
Graph data within this collection:
This collection does not contain any graph data.
Upload new data to this collection:
Track
FusionFrameExample.xml
Upload a 3D track model:
Model
Filters
Search
0 results
No graph data has been loaded
Track History
-24sec
24sec
1x
Jan 1 1970
06:21:48 UTC

Uploading abstract graph data follows a similar workflow but the data must be formatted in a Sage file. When the Sage file is uploaded, it will appear under the "Graph data" section of the collection. In order to view the graph data, the user needs to click on one of the tracks on the globe. There will be more details about this further in the manual.

Finally, a user may upload a model to a collection. This model will replace the circles on the tracks which represent the current location of the object being tracked. The model is uploaded using the upload controls directly beneath the data upload controls. The model being uploaded must be in the GLTF format. It is easy to convert other model formats to GLTF; converters to do so can easily be found online.

Interacting with Uploaded Data

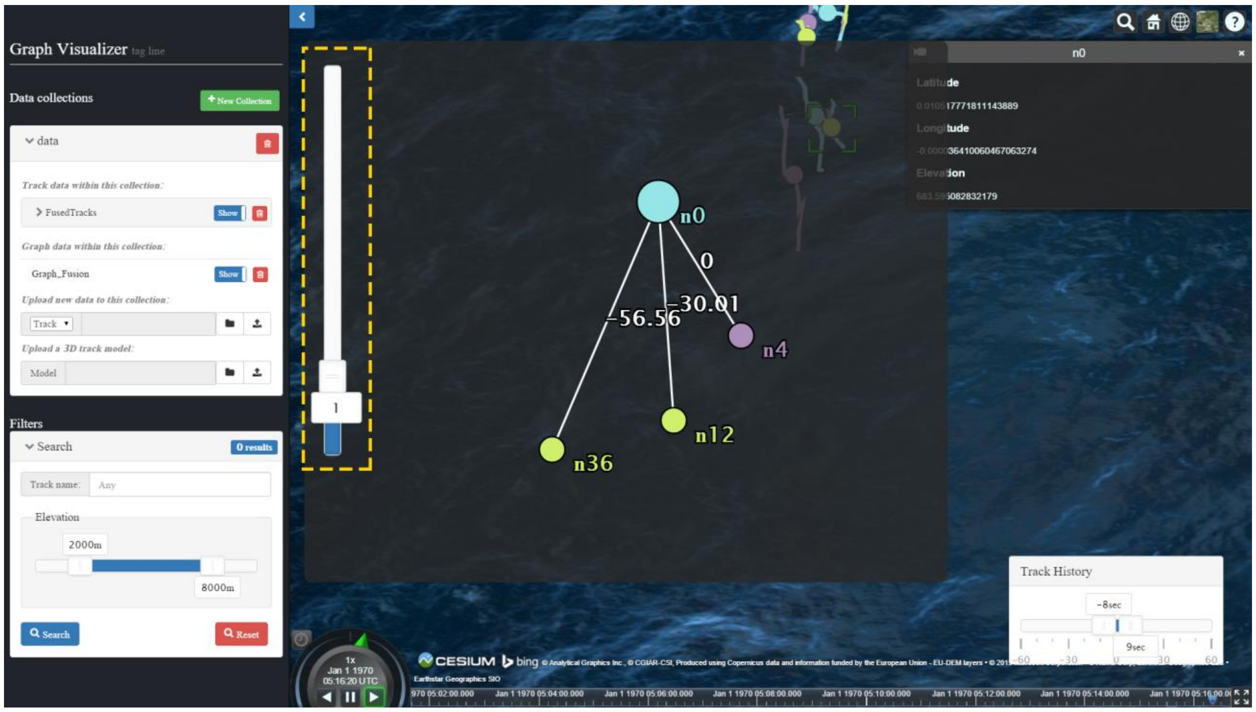
Now that the user has uploaded data, there are many ways that the user may interact with the data. The first one is that the track data within the collection is expandable by clicking on the filename of the uploaded track data. When expanded, the name of every track in the data file is listed and a toggle button is placed next to the names. This toggle button, which also appears next to the filename, allows the user to toggle the visibility of each individual track, or every track in that data file if the button next to the filename is used.



No graph data has been loaded

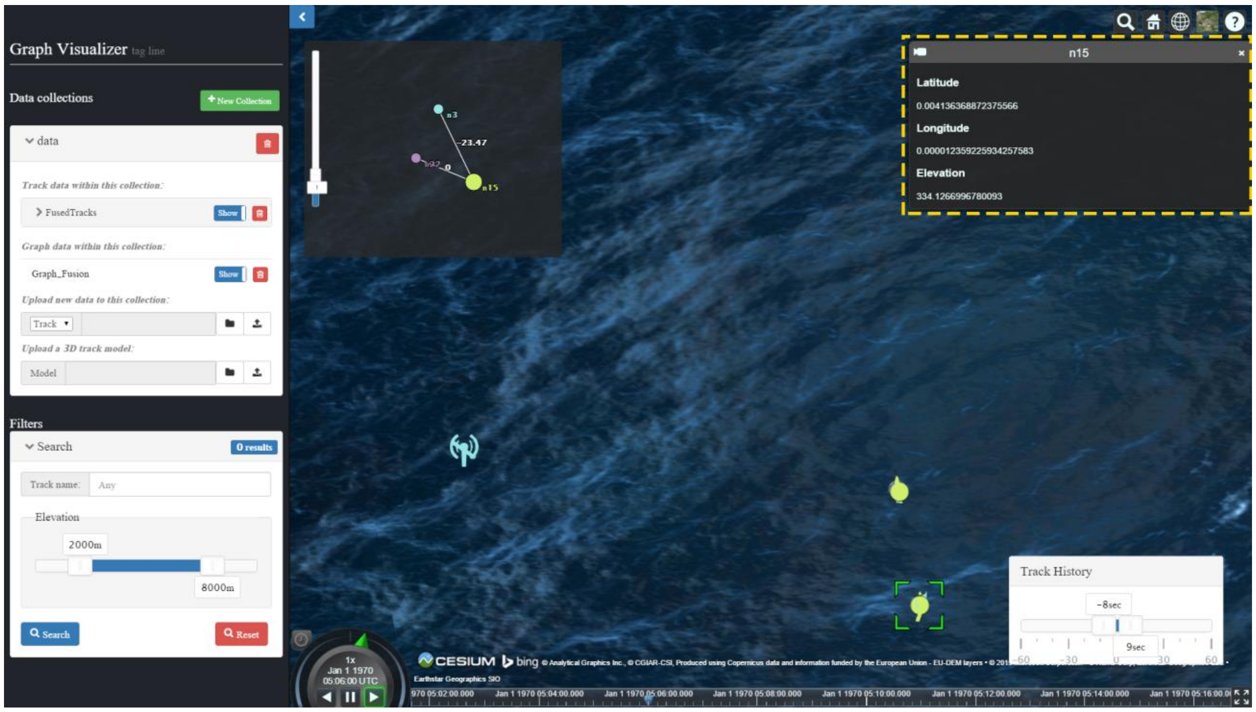
n22	Off
n4	Off
n6	On
n20	Off
n0	On
n1	Off
n2	Off
n3	Off
n15	On
n12	Off
n13	On
n14	On
n29	On
n28	On
n34	On
n37	On

Next, the user may click on one of the tracks on the globe in order to display an abstract graph centered on that track. The abstract graph will have a parent node corresponding the the track that was clicked on. The graph will initially show nodes only show nodes representing tracks that are adjacent to the parent node. The slider on the left side of the graph allows the user to control how many hops away from the parent node are displayed. For example, if the slider is at 2, all nodes that are either adjacent to the parent node or adjacent to nodes adjacent to the parent node are displayed.



Once the abstract graph is being displayed, the user may click on one of the nodes of the graph in order to make the camera lock onto the object that the node is representing and make the camera follow it. The user may also click and drag any of the nodes in order to move them around. If the user clicks and drags the background, the user may pan through the graph in order to center an important part of the graph. The graph can also be zoomed in by using the scroll wheel.

Additionally, when a track is clicked on, a window appears in the top right of the screen. This window displays the current latitude, longitude, and elevation of the object being tracked. The name of the track is displayed at the top of the window. The camera icon in the window may be used in order to make the user’s camera follow the object. The x icon in the top right of the window, closes the window.



n15

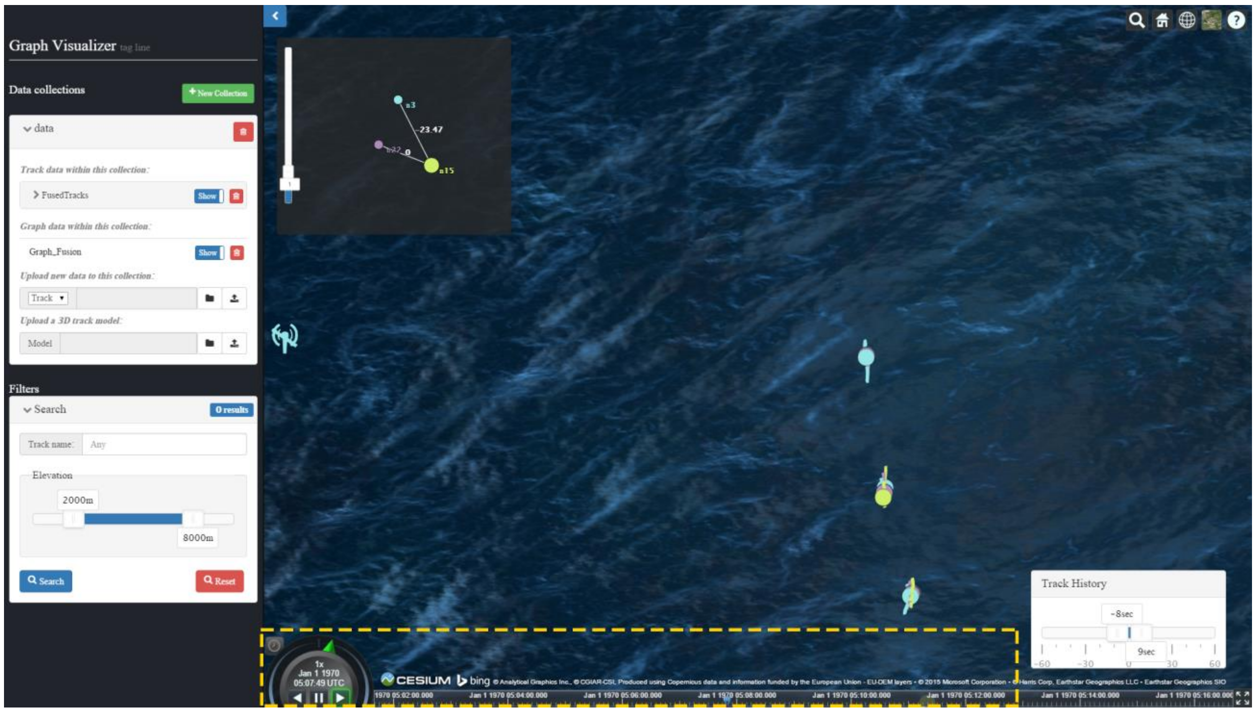
Latitude
0.004136368872375566

Longitude
0.000012359225934257583

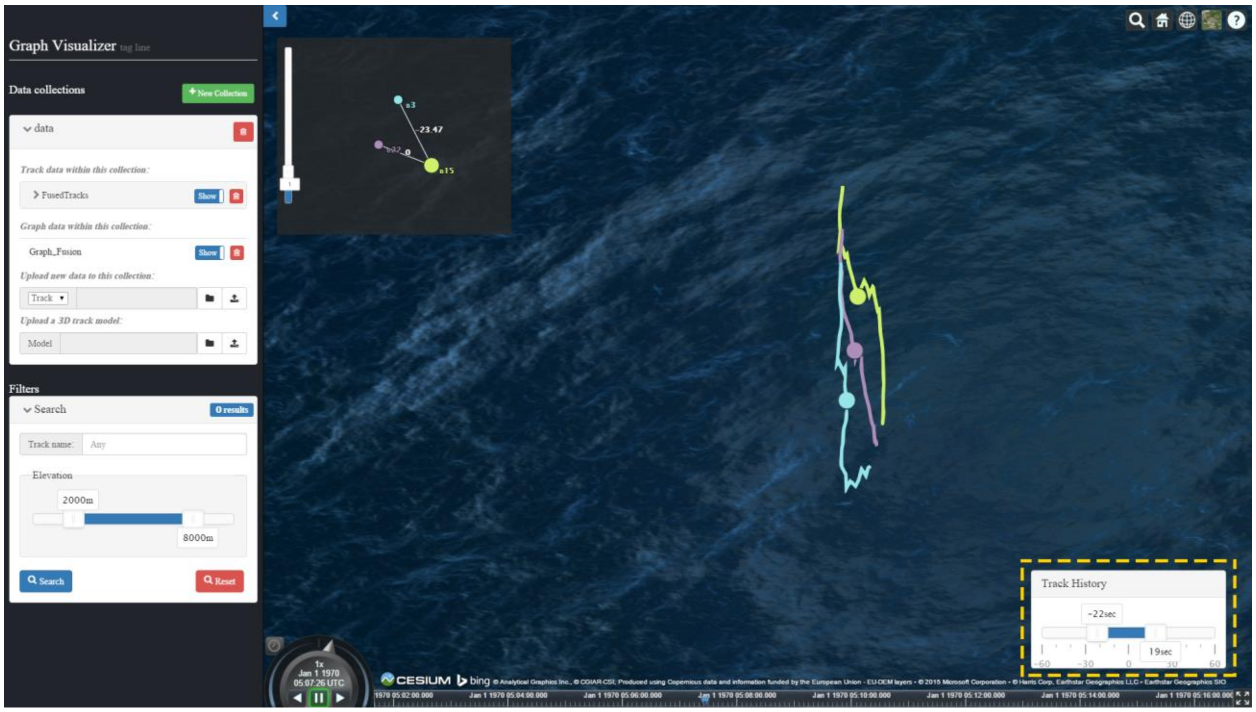
Elevation
334.1266996780093

Track Replay Controls

At the bottom center and bottom right of the screen are controls relevant to time. In the bottom center is a slider that controls the time currently being viewed. It may be clicked to change the time currently being viewed. To the left of this is a small set of controls. The controls have a green triangle; this triangle may be moved around this set of controls in order to control the rate and direction in which time is flowing. In this set of controls, there are also buttons for play, pause, and rewind, represented by a triangle point right, 2 parallel bars, and a triangle pointing left, respectively.

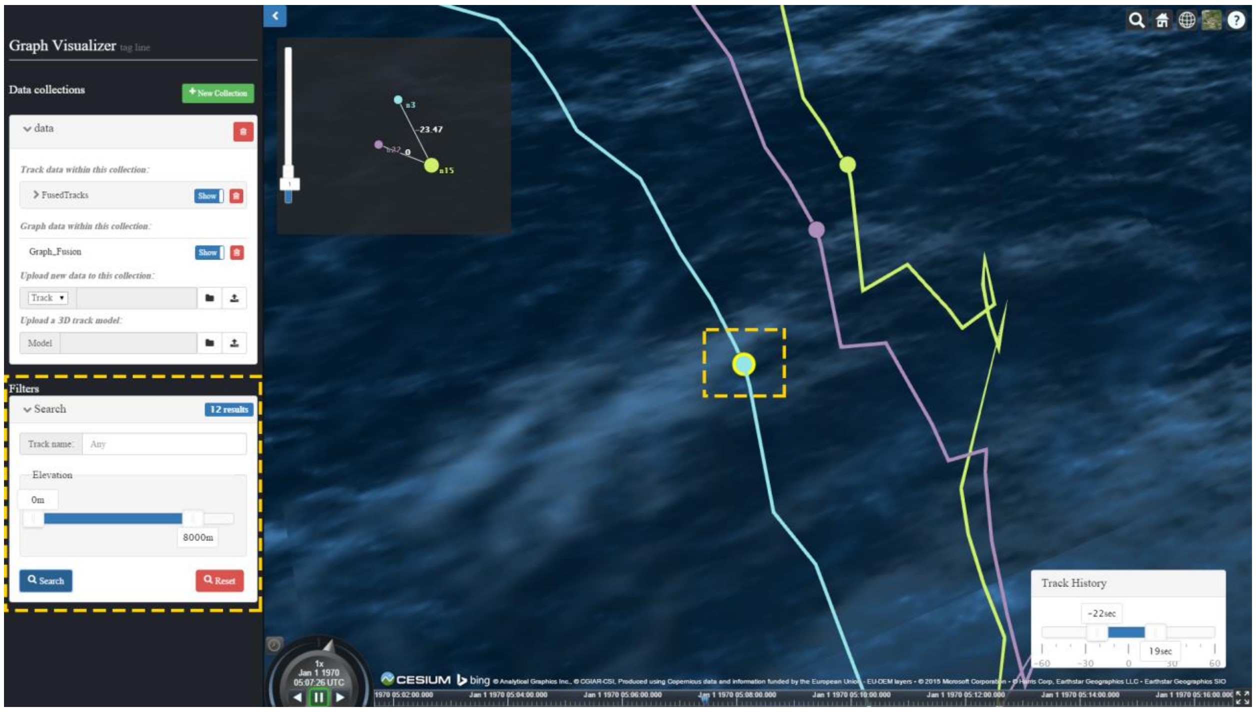


In the bottom right of the screen, there are controls in order to adjust what is referred to as the “Track History.” The history allows the user to show more or less of the objects’ locations (i.e. the lines extending from either side of the objects’ current locations). This is done by clicking on either handle on the Track History slider and moving it. Moving a handle towards the center will shorten the amount being displayed whereas moving a handle away from the center will show more.



Search

At the bottom of the controls sidebar, there are controls for searching through nodes. There are 2 ways that a user may search: by track name and by elevation. A user may type the name, or beginning of a name, into the “Track name” text field. Once a name or partial name has been typed, the user must click the button labeled “Search” in order to search through the data. The data that matches the search will be given a yellow highlight around its current location on the globe. Another way that the user may search through the data is by filtering the data by elevation. The user simply uses the slider to select the minimum and maximum elevations then clicks on the search button in order to highlight track nodes that fit the elevation criteria. When the track name and elevation are used in conjunction, only the data that fits both of the criteria are highlighted.



X. Developer Manual

In this section, we offer detailed information regarding the software and its architecture. The developer's manual provides step-by-step instructions for expanding the feature set of the software. The manual assumes that the developer is knowledgeable in JavaScript and has experience working with JavaScript prototypes, common third party libraries such as JQuery, D3.js, and Node.js. The code snippets provided below also make frequent use of the Cesium JS API. The documentation for this library is available at the following URL:

https://cesiumjs.org/Cesium/Build/Documentation/index.html

Architectural Overview

TrackWiz uses a traditional client-server architecture. The server is built through the use of Node.js. The architecture of TrackWiz is split into two main sections: the back-end and the front-end. The back-end contains functionality that is executed and managed by the server such as file upload and data storage. All back-end code is contained in the “api” folder under the root of the application. The front-end contains functionality that is executed by clients interacting with TrackWiz such as rendering of the data and viewer. All front end code is contained in the “js” folder under the root of the application.

Making Adjustments to the Server

The code that creates and manages the server is defined in app.js. This server is created through the API provided by Node.js. Any adjustments to how TrackWiz is run should be made through this file. The following snippet within app.js shows how the server is created:

```
  var server = app.listen(argv.port, argv.public ? undefined : 'localhost', function() {
    if (argv.public) {
      console.log('Cesium development server running publicly.  Connect to
http://localhost:%d/', server.address().port);
    } else {
      console.log('Cesium development server running locally.  Connect to
http://localhost:%d/', server.address().port);
    }
  });
```

HTML Templating

Each of the major components and subcomponents within TrackWiz is separated into its own HTML template. Templates allow for HTML elements to be re-added and re-used throughout the TrackWiz interface. In this software, templating is done through the use of the third party JavaScript library *Handlebars.js*. Each template is defined in its own file within the *templates* directory under the root folder.

The following figure shows the template file for a data collection.

```
<div class="panel-group">
```

```
  <div class="panel panel-default collection-{{name}}">
    <div class="panel-heading clearfix">
      <span class="accordion" data-toggle="collapse" data-target="#collection-{{name}}">
        <h4>{{name}}</h4>
      </span>
      <span class="navbar-right">
        <button class="btn btn-sm btn-danger btn-delete" data-collection="{{name}}">
          <i class="fa fa-trash-o"></i>
        </button>
      </span>
    </div>

    <div class="panel-collapse collapse in" id="collection-{{name}}">
      <div class="panel-body">
        {{>collectionList}} {{> sourceUpload}} {{> modelUpload}}
      </div>
    </div>
  </div>
</div>
```

Clearly, the template is defined using plain HTML. However, Handlebars.js provides a wide variety of inline expressions that enable developers to populate templates with context specific information. In this example, when the template is loaded, all instances of "{{name}}" are replaced with the name of the collection that is being displayed to the user.

Templates may also reference other templates within itself. Towards the bottom of the example is the line `{{>collectionList}} {{> sourceUpload}} {{> modelUpload}}`. Each of these expressions will load the contents of the template with the given name. In order for these sub-templates to be used, they must be properly registered with Handlebars.js. Only templates that are being accessed by another template need to be registered. This step is important -- If a sub-template is not properly registered, the contents of the template will not display. To register a new sub-template, a line needs to be added to the `registerAllPartials()` method within js/templates.js. This method looks like the following:

```
function registerAllPartials() {
  registerExternalPartial("datepicker");
  registerExternalPartial("sourceUpload");
  registerExternalPartial("modelUpload");
  registerExternalPartial("collectionList");
  registerExternalPartial("trackList");
  registerExternalPartial("graphList");
}
```

The registerAllPartials() method makes many calls to the registerExternalPartial(name) method. This method, as the name suggests, is registers a template as a valid sub-template within Handlebars.js. In order to make a template usable within another template, all the developer needs to do is add another call to registerExternalPartial(name) and pass in the filename of the sub-template (omit the file extension).

Creating and Applying a New Template

Creating new template for a component is a simple process. First, the developer must create a new file for the template in the templates folder. All templates must have the .hbs extension to be recognized. Within this template file, simply add the static HTML that needs to be rendered. Nextly, insert placeholders within the template file for any dynamic content that needs to be displayed within the component. To render the template onto the page, first use the getTemplateHTML() method and pass in the name of the template. This method returns a string containing the contents of the template file. Because this method runs asynchronously, the result should be handled using a JQuery promise (typically ".done()"). Inside the promise callback, use the applyTemplate() method and pass in the template string and a javascript object containing the values for each placeholder. The applyTemplate() method will return a string containing the template HTML with the dynamic placeholders properly substituted. Finally, use JQuery to append or prepend the templated string to the document. Below is a code snippet demonstrating this process. This snippet is used to append a collection to the control panel.

```
Collection.prototype.renderCollection = function(div) {
  var outerScope = this;
  getTemplateHTML('dataCollection').done(function(data) {
    var templated = applyTemplate(data, outerScope);
    var target = $(templated).prependTo(div);
  });
  this.loadCollection();
};
```

In this example, the `'dataCollection'` template is requested. The result is stored in the ‘data’ parameter within the callback function. The ‘`Collection`’ object (used as ‘`outerScope`’) contains the values of all dynamic content to be added to the collection. Passing the collection object and the raw template string into the `applyTemplate()` method will yield a templated string. Finally, the result is prepended to the ‘`div`’ object that gets passed into `renderCollection()`.

Understanding how to make proper use of templates is a key part of adding interactive controls and features to TrackWiz.

Representing New Data Source Types

Currently, this visualizer is capable of representing track and sensor data in the Cesium viewer. However, we acknowledge that additional types may need to be implemented in the future and we have designed the software so that adding new data types is trivial.

Before new data can be drawn in the Cesium viewer, an object prototype must be created to represent the data. Existing data types are implemented in js/lib/datasources. All datasources inherit methods from the “DataSource” object prototype defined in js/lib/datasources/dataSource.js. This prototype contains many of the methods that new data types will need, so there is no need to implement the methods defined in this prototype unless the developer is defining new behavior.

In this chapter, we will provide a brief overview of the DataSource prototype, and then show how to create a prototype for a new data type.

The parent datasource prototype object contains getters and setters for the following attributes:

name	the name of the track, given as a string
entities	an array of Cesium entities that are contained in the data source
clock	a Cesium clock object that controls the time window that the data occurs in
changed	a Cesium event that is triggered whenever data is added or modified within the data source
error	a Cesium event that is triggered whenever the data source encounters an error
isLoading	a boolean that tracks whether or not the data source is in a loading state
trackNode	an entity representing the location of the track as time passes
positionProp	a property that keeps track of each sample of data to help with the rendering of the trackNode
platform	the platform that the sensor that collected the track was placed on
sensorType	the type of sensor that collected the data
color	a Cesium color object representing the color of the track

Additionally, the following **public** methods are made available by the DataSource prototype:

createTrackNode()	creates a node in the TrackWiz visualizer that represents the location of the tracked entity as it moves through space. Additional data source prototypes that wish to adjust the appearance of the track node should overload this method.
setTrackModel(location)	sets the model of the track node to the GLTF file at the filepath string specified by *location*.
highlightOnCondition(callback)	highlights the track node with a yellow outline if the callback method returns true
getCovarArray(covariance)	given the covariance as a string, format the covariances as an array of values and return the result
formatCovariance(covariance)	given the covariance as a string, nicely format the covariance as a HTML string so that it can be displayed to the user.
formatTrackNodeDesc()	formats the latitude, longitude, and elevation of the track node as an HTML string so that it can be displayed to the user.

trackColor(key)	a static method that generates a unique pastel color based on the provided key. The resulting color is turned as a hex color string.

In order to add support for a new track type, a new prototype must be defined that inherits from the base DataSource prototype. The following snippet demonstrates how this can be accomplished:

```
function TrackDataSource(platform, sensorType, name) {
  DataSource.call(this, platform, sensorType, name);
}
TrackDataSource.prototype = Object.create(DataSource.prototype);
TrackDataSource.prototype.constructor = TrackDataSource;
```

First, the constructor for the new type is defined. Within the constructor, make a call to the parent Datasource constructor and pass in appropriate values. Next, the prototype field of the new type must be set to an instance of the parent DataSource. Finally, the constructor of the new data type must be set to the previously defined constructor.

Once the data type has been created, it is now necessary to define a method that handles the addition of new data to the data source. As an example, the following snippet defines a method `addStateEstimate(se, time)` that consumes an XML state estimate and time tags and adds the data to a TrackDataSource object.

```
/**
 * Adds a time and position dependant sample to the data source
 */
TrackDataSource.prototype.addStateEstimate = function(se, time) {
  var kse = se.getElementsByTagNameNS('*', 'kse')[0];
  var p = Collection.parsePos(kse);
  var covariance = kse.getAttribute('covariance');
  var formattedCovariance = this.formatCovariance(covariance);
  var position = Cesium.Cartesian3.fromDegrees(p.lat, p.lon, p.hae);

  var epoch = Cesium.JulianDate.fromIso8601('1970-01-01T00:00:00');
  var set_time = Cesium.JulianDate.addSeconds(epoch, time, new Cesium.JulianDate());

  this._positionProp.addSample(set_time, position);
  this._slideTimeWindow(set_time);
};
```

Adding and Managing HTTP Routes

TrackWiz makes extensive use of HTTP GET, POST, DELETE and PUT requests to facilitate the transfer of information between the back end and the front end of the codebase. All routes are defined in *app/routes.js*. Defining new routes is made especially easy through the use of Node.js and the Express web framework. **All routes defined in the routes.js module use "/collections" as a base url.**

At the top of the routes module a new router is defined, and the bodyParser package is used to allow us to extract url encoded parameters from the requests:

```
var router = express.Router();

router.use(bodyParser.urlencoded({
  extended: false
}));
router.use(bodyParser.json());
```

New routes are added to the router object through the use of the get, post, delete, and put methods provided by the Express router object:

```
//GET a collection with the given name
router.get('/:collectionName', function(req, res) {
  var collectionName = req.params.collectionName.replace(/\W/g, '');
  var collection = collectionSet.get(collectionName);

  if (collection === undefined) {
    res.status(404).send();
  } else {
    res.json(collection);
  }
});
```

In this code snippet, we define a new rule for GET requests to "/collections/collectionName", where collectionName is a parameter referencing the name of a collection within TrackWiz. The `req` parameter of the function is an object representing the HTTP request, and `res` represents the response that will be returned to the client. In this example, we get the collectionName parameter from the request object and strip out any illegal characters. We then retrieve the collection and insert the result into the response. Once this method is defined, a GET request issued to "/collections/foo" will retrieve the "foo" collection and return it to the client issuing the request. In the event that the collection is not found, we return a 404 NOT FOUND response to the user. Following this example, it should be fairly easy to implement new routes and extend existing ones.

File Upload Management

File upload is one of the most important pieces of TrackWiz. Without it, it would be impossible for users to provide their own data for the visualizer to represent. Files upload is accomplished through the use of a Node.js package called Multer. While multer greatly facilitates the upload process, it is still up to the developer to define how uploaded files are handled. To do so, we have defined a storage engine located in *api/storageEngine.js.* The storage engine is responsible for defining a number of characteristics about the uploaded file, such as (but not limited to):

- The name of the uploaded file
- The location that the uploaded file is stored to
- How the file should be managed (should it be stored on disk? Saved purely in RAM?)
- How a file should be removed

If future development of TrackWiz requires specific data types to be handled in presently unsupported ways, the current storage engine can be easily extended to meet these requirements.

Currently, the storage engine processes files through the use of the `_handleFile(req, file, cb)` method:

```
DataStorage.prototype._handleFile = function _handleFile(req, file, cb) {
  var uploadType = req.body.uploadType;

  if (uploadType === 'sage') {
    this._writeSageFile(req, file, cb);
  } else {
    this._writeXMLFile(req, file, cb);
  }
};
```

If the file being uploaded is a sage file (which is used for representing abstract graph data), then the storage engine defers to the _writeSageFile() method which is responsible for parsing the Sage graph file. Otherwise, the storage engine defers to the _writeXMLFile() method, which simply writes the contents of the file to disk. This distinction must be made because Sage files are not immediately interpretable by the visualizer. The _writeSageFile() method is defined as the following:

```
DataStorage.prototype._writeSageFile = function(req, file, cb) {
  var that = this;

  that.getDestination(req, file, function(err, destination) {
    if (err) return cb(err);

    that.getFilename(req, file, function(err, filename) {
      if (err) return cb(err);

      file.stream.pipe(concat(function(data) {
        var stringData = data.toString();
        var graph = Graph.fromSage(stringData);
        graph.writeJSON(destination, filename);
        cb(null, {
          destination: destination,
          filename: filename,
          path: path.join(destination,  filename),
        });
      }));
    });
  });
};
```

First, the storage engine decides what filename to give the Sage file, and where the file should be saved. Once decided, the storage engine reads the contents of the sage file and converts the byte stream to a string. Then using the graph parser, the sage file is converted to a JSON string and the resulting string is saved to disk in a file.

The storage engine must also define how uploaded files are removed:

```
DataStorage.prototype._removeFile = function _removeFile(req, file, cb) {
    var path = file.path;
    delete file.destination;
    delete file.filename;
    delete file.path;
    fs.unlink(path, cb);
};
```

Using the Storage Engine

Once the storage engine has been defined, it can be used when files are uploaded to the server. This is done in the routes module. First, a new instantiation of the storage engine must be created:

```
//Define rules for storing data sources
var diskStorage = StorageEngine({
  destination: function(req, file, cb) {
    cb(null, getDestination(req, file));
  },
  filename: function(req, file, cb) {
    var parsed = file.originalname.replace(/\.[^/.]+$/, "");
    var count = 0;
    var dest = getDestination(req, file);
    var finalName = parsed;
    while (fs.existsSync(path.join(dest, finalName))) {
      finalName = path.join(parsed + "_" + (++count));
    }
    cb(null, finalName);
  }
});
```

As parameters to the storage engine, we provide an object that offers suggestions to the storage engine regarding where to store the file and what to name the file. However, the storage engine is under no obligation to take these suggestions. The storage engine can, for example, choose to store the file in a temporary location if the suggested destination is invalid or nonexistent. The logic to make these decisions in defined in the storage engine itself.

Once we have instantiated a storage engine, we must then create a Multer upload handler:

```
//Define a handler for uploading data source files
var datahandler = multer({
  storage: diskStorage,
  limits: {
    fileSize: 1024 * 1024 * 100
  },
  fileFilter: function(req, file, cb) {
    var accepted = ['text/xml', 'application/octet-stream'];
    if (accepted.indexOf(file.mimetype) !== -1) {
      cb(null, true);
    } else {
      cb(null, false);
    }
  }
}).single('file');
```

The multer upload handler takes in an object that defines how it operates. Here, we declare that we are going to use the previously instantiated storage engine to handle the storage of uploaded files. We then include a “limits” parameter that defines a maximum file upload size of 100 megabytes. Finally, we include a filter parameter to define what types of files are considered valid. In this case, we only accept files that have an approved mimetype.

With our upload handler properly defined, we can then bind the handler to listen for POST requests to “/collections/upload/data/”:

```
//Bind post to the handler, and catch errors
router.post('/upload/data', function(req, res) {
  datahandler(req, res, function(err) {
    //Something went wrong
    if (err) {
      res.status(500).send("Failed to upload file: " + err);
      return;
    }
    //Upload was successful
    if (req.file === undefined) {
      res.status(400).send("No file data specified.");
      return;
    }
    var collectionName = req.body.collectionName.replace(/\W/g, '');
    var collection = collectionSet.get(collectionName);
    var uploadType = req.body.uploadType;

    var uploadInfo = {
      destination: req.file.destination,
      filename: req.file.filename,
      mimetype: req.file.mimetype,
      uploadType: uploadType
    };

    res.json({
      context: collection,
      file: uploadInfo
    });
  });
});
```

Parsing and Loading Graph Data

Before data can be added to TrackWiz, it must be accessible in a format that is easily parsed. JavaScript already offers methods to interpret XML and JSON data so it is recommended to input data into TrackWiz in either of these formats. However, in cases where other data formats must be supported, it is possible to convert the data for easier access. We have done so in this project to add support for the Sage file format. Sage files are used to represent nodes and edges, and associations of entities within the abstract graph. The code that is responsible for making the conversion from Sage to JSON is available in api/Graph.js. This module is executed by the storage engine whenever a Sage file is uploaded to the server. The converted JSON output is then saved to disk and accessed by the visualizer. In this way, TrackWiz is able to convert the following Sage data:

```
G=DiGraph(weighted=true,sparse=True)
G.add_vertices([\
'34',\
'22',\
'4',\
'45',\
'36',\
'0',\
'1',\
'2',\
'15',\
'37',\
'6',\
'12',\
'13',\
'20',\
'28',\
'29',\
'3',\
'14',\
'33',\
])
G.add_edges([\
('15', '22', '0.000e+00'),\
('3', '22', '0.000e+00'),\
('45', '29', '0.000e+00'),\
('1', '13', '-9.630e+01'),\
('2', '14', '-2.087e+01'),\
('1', '6', '0.000e+00'),\
('0', '12', '-3.001e+01'),\
('28', '45', '-6.125e+01'),\
('28', '29', '0.000e+00'),\
('15', '3', '-2.347e+01'),\
('0', '4', '0.000e+00'),\
('2', '20', '0.000e+00'),\
('0', '36', '-5.656e+01'),\
('13', '6', '0.000e+00'),\
('12', '4', '0.000e+00'),\
('36', '37', '0.000e+00'),\
('14', '20', '0.000e+00'),\
('33', '34', '0.000e+00'),\
])
```

Into the following JSON:

```
{
  "vertices": [
    {
      "id": "n34"
    },
    {
      "id": "n22"
    },
    {
      "id": "n4"
    },
    {
      "id": "n45"
    },
    {
      "id": "n36"
    },
    {
      "id": "n0"
    },
    {
      "id": "n1"
    },
    {
      "id": "n2"
    },
    {
      "id": "n15"
    },
    {
      "id": "n37"
    },
    {
      "id": "n6"
    },
    {
      "id": "n12"
    },
    {
      "id": "n13"
    },
    {
      "id": "n20"
    },
    {
      "id": "n28"
    },
    {
      "id": "n29"
    },
    {
      "id": "n3"
    },
    {
      "id": "n14"
    },
    {
      "id": "n33"
    }
  ],
  "edges": [
    {
      "source": "n15",
      "target": "n22",
      "weight": 0
    },
    {
      "source": "n3",
      "target": "n22",
      "weight": 0
    },
    {
      "source": "n45",
      "target": "n29",
      "weight": 0
    },
    {
      "source": "n1",
      "target": "n13",
      "weight": -96.3
    },
    {
      "source": "n2",
      "target": "n14",
      "weight": -20.87
    },
    {
      "source": "n1",
      "target": "n6",
      "weight": 0
    },
    {
      "source": "n0",
      "target": "n12",
      "weight": -30.01
    },
    {
      "source": "n28",
      "target": "n45",
      "weight": -61.25
    },
    {
      "source": "n28",
      "target": "n29",
      "weight": 0
    },
    {
      "source": "n15",
      "target": "n3",
      "weight": -23.47
    },
    {
      "source": "n0",
      "target": "n4",
      "weight": 0
    },
    {
      "source": "n2",
      "target": "n20",
      "weight": 0
    },
    {
      "source": "n0",
      "target": "n36",
      "weight": -56.56
    },
    {
      "source": "n13",
      "target": "n6",
      "weight": 0
    },
    {
      "source": "n12",
      "target": "n4",
      "weight": 0
    },
    {
      "source": "n36",
      "target": "n37",
      "weight": 0
    },
    {
      "source": "n14",
      "target": "n20",
      "weight": 0
    },
    {
      "source": "n33",
      "target": "n34",
      "weight": 0
    }
  ]
}
```

Once the data has been converted, in can then be loaded into TrackWiz. The method that handles the loading of graph data can be found in js/chart.js:

```
D3Graph.prototype.loadGraphFile = function(filePath) {
  var outerScope = this;
  var newEdge = {},
    newVert = {};
  var vertList = this.fullGraph.vertices;
  var edgeList = this.fullGraph.edges;
  var d = $.Deferred();

  d3.json(filePath, function(error, graph) {
    if (error) {
      throw error;
    }
    for (var i = 0; i < graph.vertices.length; i++) {
      outerScope.addVertice(graph.vertices[i], vertList);
    }
    for (var j = 0; j < graph.edges.length; j++) {
      outerScope.addEdge(graph.edges[j], edgeList, vertList);
    }
    outerScope.addOrphanEdges();
    outerScope._start();
    d.resolve();
  });
  return d;
};
```

This method begins by retrieving a list of edges and vertices that have already been loaded into the abstract graph. Then, we use the D3 api to asynchronously read the converted JSON graph file. Then, each of the new vertices and edges are loaded into the graph through the use of `addEdge()` and `addVertice()`. If any of the edges within the graph reference a node ID that does not exist, the edge is added to a list called "`orphanEdges`". Any time a new graph file is loaded, the list of orphan edges is checked again in case the addition of the new file has allowed any of the previously unresolvable edges to be successfully loaded into the graph.

XI. Bibliography

[1] Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Upper Saddle River: Prentice-Hall, 2010. Print.

[2] "Cesium - WebGL Virtual Globe and Map Engine." *Cesium*. N.p., n.d. Web. 21 Sept. 2015.

[3] "D3.js - Data-Driven Documents." D3.js - Data-Driven Documents. N.p., n.d. Web. 21 Sept. 2015.

[4] Mitchell, H. B. Data Fusion: Concepts and Ideas. Heidelberg: Springer, 2010. Print.

[5] "A Review Of Data Fusion Techniques." A Review of Data Fusion Techniques. Web. 21 Sep. 2015. <http://www.hindawi.com/journals/tswj/2013/704504/>

[6] Burns, M., Lukens, I., & McAndrews, C. (2014, March 17). Tracking Testing Framework. Retrieved September, 2015.

[7] Battista, Giuseppe Di, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for Drawing Graphs: An Annotated Bibliography."*Computational Geometry* 4.5 (1994): 235-82. Web.