Tech Review

Totality AweSun

Bret Lorimore, Jacob Fenger, George Harder

November 14th, 2016

CS 461 - Fall 2016

Abstract

This document compares various technologies for use in the North American Solar Eclipse 2017 senior capstone project. Specifically, we look at three specific parts of the project, the Eclipse Image Processor, the Image Processor Manager, and the Eclipse Simulator. For each of those high level components, we review applicable technologies for three of their sub-components. For each of the nine total sub-components, we make a conclusion about which of the considered technologies is best suited to our needs, and will thus be used in our implementation.

1 INTRODUCTION

This technology review proceeds in three similarly structured sections. First, George Harder discusses which technologies will be the most effective for the implementation of the Eclipse Image Processor. This section is broken down into three subsections that correspond to the pieces he is responsible for in this project. The pieces are Image Classification and Manipulation, Runtime Environment, and Circle Detection. In the next section Bret Lorimore discusses his area of responsibility, the Image Processor Manager. The three pieces that Bret examines are Downloading/Uploading from Google Cloud Storage, Storing Photo Metadata, and Application Parallelization. Lastly, Jacob Fenger details the pieces and technologies that will comprise the Eclipse Simulator. The three pieces he will be responsible for are User Interface, Gathering User Latitude/Longitude Information, and Displaying Sun/Moon Based On Latitude, Longitude, and Time Information.

An important note with regard to some of the technologies being discussed is that Google is sponsoring this project and as such we have been asked to use Google products when possible. This is especially relevant to the sections on runtime environments and storing the photo metadata.

2 ECLIPSE IMAGE PROCESSOR

The eclipse image processor is the piece of our project that handles the spatial and temporal alignment of the eclipse images that are uploaded to the Eclipse Megamovie website. We have identified three pieces into which the image processor can be broken down. These are image classification and manipulation, the runtime environment, and circle detection. In order for this element of our project to operate effectively it is critical that these three pieces utilize robust and functional technologies. This section of the technology review details what options are available for implementing each of these three pieces, analyzes these options, and arrives at a determination as to which option is the best in the context of this project.

2.1 Image Classification and Manipulation

The image classification and manipulation portion of the eclipse image processor encapsulates a few key requirements for this project. For our project to be functionally complete it must determine whether the sun/moon are in a state of totality, align them spatially, and crop them such that the sun is the same size in all of the images. This process requires the use of a computer vision application programming interface (API). Three possible options of APIs are: OpenCV, SimpleCV, and VXL.

The relative strengths and weaknesses of these three options can be evaluated using several key criteria that emerge when considering the requirements for this element of the project. These criteria are: speed, support/documentation, availability, ease of installation, and ease of use. Fortunately, all three of these tools are available online and provide documentation on installation so these categories do not require much discussion

[1]–[3]. With these criteria met, and considering the performance requirements of our project, the next most important criteria to consider is speed.

OpenCV and VXL both rank highly in the speed category because they are written natively in C++ [1], [2]. SimpleCV on the other hand is written in Python and thus uses an interpreter [3], which causes speed penalties. SimpleCV's use of Python is a significant drawback and it cannot be ignored. Because of the large volume of images this tool is expected to process, we are targeting a mean processing time of 1 second per image. This is not necessarily impossible with an interpreted language, but it introduces major difficulties in trying to meet this requirement. OpenCV and VXL are the clear winners when considering speed.

While not as critical to meeting specific requirements as speed, it is important to consider the costs associated with learning a new API. While all three APIs provide online documentation, these docs are by no means equal. OpenCV and Simple CV both provide easy to understand and what appear to be comprehensive tutorials [1], [3]. While OpenCV may be more complicated than the aptly named SimpleCV it has a massive number of contributors on question sites like Stack Overflow [4]. Having an online community as a resource is an invaluable asset when learning an API and this is one of the great strengths of OpenCV. VXL, unlike the other two possibilities provides relatively poor and hard to navigate documentation. The time required to learn how to use an API can significantly slow down the development process of an application like our image processor. Having high quality documentation and tutorials at our disposal, as is the case with OpenCV and SimpleCV, is critical to surmounting the learning curve and its importance should not be overlooked.

TABLE 1
Comparison of Possible Technologies for Image Classification and Manipulation

	Available	Installation	Speed	Support and Documentation	Ease of Use
OpenCV	Yes	Manageable	Fast (native C++)	Excellent. Tutorial, docs, huge online community	Medium. May take some learning, offset by support/ documentation
SimpleCV	Yes	Manageable	Slow (native Python)	Good. Tutorial, docs, book available for purchase	Easy. Meant to be simple by design.
VXL	Yes	Manageable	Fast (native C++)	Poor. Hard to read docs, not much support online.	Difficult. Collection of libraries, poor docs, likely tough to learn.

Conclusion [See summary of previous discussion in Table 1]: OpenCV and SimpleCV are highly comparable

in terms of ease of use and support/documentation. SimpleCV may be easier at first, but any advantage it has is offset by the robust online community willing to support each other and answer questions that may arise when using OpenCV. VXL falls flat when compared to OpenCV and SimpleCV in those two categories. With this in mind, and when considering that speed is of critical importance, OpenCV is the clear choice of computer vision API for this project.

2.2 Runtime Environment

The high volume of images the Eclipse Megamovie project expects to collect necessitates some form of scalability for our image processor. Without a runtime environment that we can effectively scale to process hundreds of thousands or perhaps millions of images in a timeframe that meets the requirements of the larger system, the image processor's utility is near zero. We have identified three potential options to meet the needs for scaling this image processor: Docker containers, raw cloud based virtual machines (VMs), and Google Cloud Functions (GCF).

In order to determine which runtime environment will best allow us achieve the scalability necessary for the eclipse image processor several criteria will be considered. In this case they are: availability, cost, integration with our application, security, and overhead. Our group's partnership with Google for this project produces a natural inclination toward using the Google Cloud Platform for whichever of these options we end up using. Thus, all of these products have equal availability and our sponsor has made it clear that any monetary costs associated with using Google products are negligible. Because all of the options rank equally in the availability and cost categories only the other criteria will be discussed below.

As the Google Cloud Platform supports all three possible technologies we are considering, integration with our eclipse image processor would appear to be essentially equal across all three options. However, this is not the case. Using cloud based VMs or Docker Containers allows us to encapsulate our application in a manner that allows for rapid scaling in addition to easy retrieval and transmission of images and data [5], [6]. GCF on the other hand uses an event based microservice approach to app deployment [6]. Instead of containing our application, GCF is a Javascript function that would call our eclipse image processor whenever an event is triggered [6]. In this case the event would be a photo upload to the Eclipse Megamovie website. Rather than encapsulating our application GCF would spawn a child process, the eclipse image processor's executable, each time it was called. This results in a more complex deployment strategy because we would need to determine how to retrieve, store, and transmit data without access to a file system.

Security is of paramount importance when deploying a web based application that will be handling a high volume of uploads and downloads. User information being compromised as the result of a vulnerability in our application would inevitably harm participation in any future projects similar to this one. Fortunately, the Google Cloud Platform has a strong commitment to security [6]. Google Compute Engine VMs encrypt all

of their data as it is stored and transmitted [6]. This is an enormous advantage in terms of security because it allows us to focus on developing our application's functionality rather than worrying about securing it. Docker containers are deployed on top of VMs, so they have all of the same security advantages as a Compute Engine VM. In addition to this, Docker containers are completely isolated from the rest of the VM they are running on [5], so if one were compromised the issue could be handled in a manner that does not impact any other containers. GCF has its own advantages in terms of security. Each function runs in its own execution environment [5]. Like the Docker containers, this means that all of the instances of GCF are separate from one another. GCF is also running in what Google calls a "serverless" environment [5], this means that GCF is not storing data or using hardware that could be compromised. All of the options receive high marks in the security category.

In the context of evaluating these options, overhead refers to the amount of infrastructure that would need to be maintained in order to support each option as it scales to meet the demands of a high volume of uploads. For Google Compute Engine VMs, the overhead is fairly high. Each VM requires its own guest operating system, virtual memory, and virtual hardware to run an instance of our application [5], [6]. This is in contrast to Docker containers, which can run multiple instances of an application on top of a single host [5]. Each instance of the application only needs its dependencies to run. This model requires less overhead in order to scale our application, which in this case results in fewer places where things can go wrong. By far the most lightweight approach, GCF operates using a serverless model [6]. This provides advantages to us as developers because it greatly simplifies deployment. However, it presents new challenges in terms of how we retrieve and transmit data. In addition to the challenges of data management, another disadvantage of the serverless model is that it takes away our control over how much compute power we can provide to the eclipse image processor.

Conclusion [See summary of previous discussion in Table 2]: Taking all criteria into consideration, the best choice of runtime environment for our application will be Docker containers. These retain the best elements of the other options while shedding their defects.

2.3 Circle Detection

Detecting circles is an area of ongoing research in computer science and is of particular interest to this project because it is necessary for the temporal ordering and spatial alignment of the eclipse images. In order to determine the temporal ordering of the images we plan to use the relative difference in position of the centers of the sun and the moon. To spatially align the images we will be rotating them so that the visible portion of the sun is at a fixed angle relative to the center of the image. For either of these processes to be successful we need a technique capable of locating circles and their centers. We have identified three possible algorithms, Hough transforms, Blob detection, and the Ant System Algorithm.

TABLE 2
Comparison of Possible Technologies for Runtime Environments

	Available	Cost	Integration	Security	Overhead
Virtual Machines	Yes	Neglible	Good, can encapsulate app and dependencies easily	Excellent	High, VM per instance has high cost
Docker Containers	Yes	Neglible	Good, provides encapsulation	Excellent	Medium, clusters deployed on top of VMs
Google Cloud Functions	Yes	Neglible	Okay, JS can call our executable but introduces unnecessary complexity	Excellent	Low, lightweight serverless Javascript function calls

In order to select an algorithm for circle detection, three primary criteria must be considered. These are computational complexity, accuracy, and implementation difficulty. Computational complexity is of high importance because a slow algorithm or one that uses a large amount of memory jeopardizes our chances of meeting performance requirements and places strain on our runtime environment. We need this algorithm to achieve high accuracy to ensure the seamless integration of the eclipse image processor with the Eclipse Megamovie project. If our algorithm cannot correctly identify circles, our processor could not be considered functional. Lastly, an important consideration is the level of difficulty required in integrating our algorithm of choice into the eclipse image processor's codebase.

The Hough transform is a common algorithm for circle and line detection that is known to have high computational complexity and memory usage [7], [8]. This is generally disadvantageous to the performance of our eclipse image processor. Blob detection has a lower computational complexity and memory usage than the Hough transform because it does not need to store an array of all of the possible circle centers like the Hough transform does [7], [9], [10]. The least computationally complex algorithm is the Ant System Algorithm described by Chattopadhyay et al. [8]. This algorithm's complexity is determined solely by the number of pixels on the edge of shapes in the image. This is significantly lower than the complexity of other two algorithms which rely on the number of pixels in the whole image.

Accuracy is of critical importance to the success of the eclipse image processor. As such, it is necessary that the circle detection algorithm consistently detects the sun and moon in the eclipse images. Hough transforms have been shown to be very successful in this regard [11]. To this group's knowledge, blob detection has not

been shown to be effective at finding the sun and moon in eclipse images. OpenCV does include a blob detector that can filter blobs by their relative circularity and return their centers [9], [10], which meets the needs of this processor. However, the accuracy of blob detection is likely to suffer in this application because the pixels corresponding to the moon are likely to blend with the background of the image since they are both black. This results in there being no distinct blob to be detected in the crescent images. The Ant System Algorithm, while shown to be accurate [8], could fall victim to this same problem. The images that the Ant System Algorithm were tested on had white outlined shapes over a black background [8], this is not how the real world eclipse images will appear.

The final consideration in selecting a circle detection algorithm is the effort it will take to use it in the eclipse image processor. Both the Hough transform and blob detection score very highly in this category. OpenCV has implementations of these methods, documentation on how to use them, and even has code examples of their use [9], [12]. This is extremely useful and makes the implementation of the circle detection portion of the image processor much simpler. The Ant System algorithm on the other hand would need to be implemented by our team and would be based on pseudocode in the Chattopadhyay et al. paper. This presents a major disadvantage because it jeopardizes both the accuracy and performance of the algorithm. There is no guarantee that our implementation of the algorithm will work as well as the one the author's of the paper used. This also requires potentially several extra weeks of development and testing. Having to implement our own version of the Ant System Algorithm essentially eliminates it from contention as a possible circle detection algorithm.

TABLE 3
Comparison of Possible Technologies for Circle Detection

	Available	Cost	Integration
Hough Transform	High, uses a lot of memory and iterates over each pixel more than once and performs a computation each step	Excellent, known to have good accuracy and has been shown to work on eclipse images	Low, supported in OpenCV
Blob Detection	Medium, iterates over each pixel once performing computations at each step	Good, known to be accurate at finding circles and centers but hasn't been known to work with eclipses	Low, supported in OpenCV
Ant System Algorithm	Low, has complexity O(edge pixels), could be compromised by our implementation	Okay, not shown to work on real world images, could be worsened by our implementation	Extreme, we would have to write our own implementation based on pseudocode

Conclusion [See summary of previous discussion in Table 3]: While the Hough transform may struggle in terms of computational complexity, it emerges as a clear favorite when accuracy and ease of implementation

are taken into account. It is the only algorithm known to successfully detect the sun and moon in eclipse images and is supported in OpenCV.

3 Image Processor Manager

The image processor manager will be a Python application responsible for collecting/downloading user uploaded eclipse images to be processed by the image processor application, invoking the image processor with these images as input, and collecting the output of the image processor application and uploading it to Google Cloud. In this section of the technology review we consider various technologies for downloading/uploading photos from Google Cloud Storage where they will be stored, several technologies to manage our photo metadata, and different methods for incorporating parallelization into this application. We will be evaluating these various technologies on the following criteria as applicable: functionality, security, speed, ease of development, and ease of integration with the rest of the project.

3.1 Downloading/Uploading from Google Cloud Storage

There are several methods we could use to download/upload images from Google Cloud Storage. The first of these is the gsutil application. gsutil is part of the Google Cloud SDK and makes it easy to list the contents of a storage bucket, download files from a bucket, upload files to a bucket, and more from the command line [13], [14]. For our purposes, authentication using service accounts would be most suitable. Service accounts can be thought of as user accounts for applications where the applications ”log in” using special authentication credentials instead of usernames/passwords. Configuring service account authentication for gsutil requires enabling a particular service account with the entire Google Cloud SDK using the gcloud application and a service account credentials file. As gsutil is a command line utility, it would need to be integrated with our app as a subprocess, using the Python `subprocess` module.

Google Cloud Storage is also accessible via a JSON Rest API. Like gsutil, this API allows users to easily list the contents of a storage bucket, download files from a bucket, upload files to a bucket, and more via HTTP requests [15]. Since our application has its own Cloud Storage buckets, authentication for the JSON API would be handled using service accounts [15]. Integrating the JSON API with our application would require use of a Python HTTP library and explicit creating of all the required API request URLs [15].

The final method we consider for interacting with Google Cloud Storage is the Google Cloud Client Library for Python. This client library is easily installable using pip (the standard Python package manager) and once installed, it allows programmatic access to Google Cloud Storage via a simple Python API [16]. Authentication for this client library is, like with the JSON API and gsutil, handled using service accounts [16]. Setting up this authentication is simple and is done by setting a special environment variable to hold the path to a credentials file, obtained from Google [16]. As this client library provides a native Python API, integrating it with out application would simply require importing the necessary modules and embedding API calls directly in the

source code via Python method calls [16].

Conclusion [See summary of the above comparison in Table 4]: The Google Cloud Client Library for Python is the best option for our needs as it is easy to install, it is secure, it provides all the required functionality, and is very simple to integrate with our application source code.

TABLE 4
Comparison of technologies for downloading/uploading from Google Cloud Storage

	Provides needed functionality?	Installation procedure	Authentication	Integration with our app	Ease of installation/ integration	Secure?
`gsutil`	Yes	`apt-get`, copy service account credentials file, setup service account as default auth method using gcloud	Service accounts	Invoke as subprocess	Medium, medium	Yes
Cloud Storage JSON API	Yes	Install Python HTTP library, copy service account credentials file	Service accounts	Make HTTP requests using some Python HTTP library	Easy, hard	Yes
Google Cloud Client Library for Python	Yes	`pip`, copy service account credentials file, set environment variable	Service accounts	Native integration into source code	Medium, easy	Yes

3.2 Storing Photo Metadata

For each photo we will store a metadata record containing various information including the filename, whether or not the photo has been processed, and if it has, output information from the image processor. The image processor manager will use these records to know which photos to request from Cloud Storage (ones that have not been processed), and these records will be updated after processing to include the output of the image processor.

We will evaluate several potential technologies to facilitate this data storage/manipulation. The first of these is Google Cloud Datastore. Datastore is a simple NoSQL database solution [17]. This means that it is schemeless and typeless, so programmers must be careful to validate that the data being inserted conforms to both the desired scheme and type. Datastore is a fully managed solution so users do not need to worry about setting up virtual machines or managing any of the other infrastructure associated with running Datastore [17]. Programmers can easily interact with Datastore via the Google Cloud Client Library for Python that is

mentioned in the previous section [18]. This library takes care of authenticating requests using service accounts [18]. The API additionally provides access to Datastore transactions [18]. This functionality is important for our purposes, as we will need to request a list of image files and mark them as pending processing before any other application can read a list containing any of these same files. This behavior is necessary to ensure that multiple nodes do not pull down the same image files to process.

The second solution we consider is Google Cloud Bigtable. Bigtable is a managed NoSQL database solution and is similar to Datastore in many ways [19]. Bigtable is tailored to applications that need to store massive amounts of data and query it very quickly [19]. As such, it is a much more heavyweight solution than Datastore. It offers users much more control than Datastore does. Bigtable allows users to create their own clusters and specify the type of hardware on which they will run [20]. Users can also create multiple tables as necessary [20]. These are features that are not offered with Datastore. As part of Google Cloud, Bigtable offers a Python API to interact with it and handle authentication using service accounts [20]. Bigtable has many of the same basic features as Datastore but offers users much more control and access to much more powerful systems. Google does not recommend Bigtable if you are working with less than 1TB of data [19]. Additionally, bigtable does not support transactions [19].

The last solution we investigate is running our own MySQL servers. This solution would serve our needs as we would be able to store all our data and MySQL does support transactions. That being said, it would be a great deal of work to use our own MySQL servers. We would have to create/manage our own virtual machines, manage database replication/backups, setup some sort of load balancer to send request to the different MySQL nodes, and configure access controls which is non-trivial to do securely. One upside of using MySQL is that it has a scheme and is typed, so this relieves some of the pressure from programmers to validate data based on scheme/type. In order to connect our Python code to MySQL, we would need to incorporate an additional MySQL connector library into our application [21].

Conclusion [See summary of the above comparison in Table 5]: Datastore will serve the needs of our project best as it is very simple to setup and manage, is simple to integrate with our application, and provides all the functionality we need. Despite not supporting transactions, Bigtable could likely be made to work, however this would just be adding more overhead in both setup and creating a transactions workaround. On top of this, Bigtable is fundamentally not the right solution for our needs. It is targeted at applications that store/query much more data than ours does.

3.3 Application Parallelization

Downloading/uploading images from Google Cloud Storage and running the image processor on a set of images are both relatively high latency operations. For that reason, it is desirable for this application to do multiple operations concurrently. Here we consider various solutions for doing this in Python. In this

TABLE 5
Comparison of technologies for storing photo metadata

	Provides needed functionality?	Setup procedure	Authentication	Integration with our app	Ease of setup	Secure?
Google Cloud Datastore	Yes	Enable Datastore in Google Cloud project	Built in: service accounts	Google Cloud Client Library for Python	Easy	Yes
Google Cloud Bigtable	Not natively, no support for transactions	Create Bigtable cluster, required tables	Built in: service accounts	Google Cloud Client Library for Python	Medium	Yes
Custom MySQL servers	Yes	Create virtual machines, install MySQL, configure network access, configure access controls, setup backup/replication credentials file, set environment variable	Requires custom setup	Requires Python-MySQL connector	Hard	Yes, if done correctly

comparison, we consider invoking the image processor binary using the builtin Python `subprocess` module. This module provides a nice Python wrapper around standard `fork`/`exec`/`waitpid`/etc. C functions.

The first solution we consider is standard multithreading using the builtin Python `threading` module. This can be used to both invoke multiple image processor processes concurrently and to make concurrent requests to Google Cloud Storage and Datastore. A common issue with use of the Python `threading` module is the Python Global Interpreter Lock (GIL) [22]. The GIL is a global lock in the CPython interpreter that allows only one thread to execute code in the interpreter at a time [22]. The GIL would not cause us many issues when invoking the image processor binary, as the Python subprocess implementation releases the GIL before calling waitpid on a particular sub-process. This means that the setting up/calling of a particular invocation of the image processor binary would be done serially, but the actual processing could be done currently as it is done outside of the Python interpreter and the GIL is released [22]. The GIL is also released when making socket calls, so while setup and pre/post-processing of Cloud Storage uploads/downloads would be serialized, the socket calls themselves could be working in parallel [22]. Implementation using the `threading` module requires explicit creation/starting/joining of `Thread` objects.

An alternative solution is to use the builtin Python `multiprocessing.Pool` class. This method sidesteps

the GIL entirely by using processes for parallelism instead of threads [23]. This method is also desirable as the API is incredibly simple - a pool of n processes can be created in a single line and then these processes can be passed a target function and data in a single other line of code [23]. The only requirement here is that all parameters passed to the process pool must be picklable [23]. This would not be an issue for us as our data will take the form of standard Python datatypes. This approach does require creating entire new processes, however on Linux this can be done very very quickly. In practice, regardless of the application, sidestepping the GIL and implementing concurrency using processes instead of threads is almost always the faster solution in Python.

The last parallelization solution we investigate is applicable only to calls to Google Cloud Datastore. This is using batch requests instead of single requests. The Datastore API has built in batch request functionality which allows multiple queries to made in a single API call, meaning only a single HTTP request is made for all grouped the queries [24]. When possible, this is desirable over multiple concurrent requests as there is reduced overhead with establishing TCP connections. This application will never be sending/receiving large pieces of data to Datastore, so the savings associated with establishing far fewer TCP connections is highly desirable.

Conclusion [See summary of the above comparison in Table 6]: The `multiprocessing.Pool` class is the best solution for us for invoking the image processor and uploading/downloading from Google Cloud Storage, as it is very simple to integrate and will almost certainly offer higher performance than using the `threading` module. However, for making calls to Datastore, batch requests are preferable to creating concurrent requests using the `multiprocessing.Pool` class as with batch requests we only have the overhead of creating a single TCP connection for all the datastore queries. So we will selectively use *both* the `multiprocessing.Pool` class and Datastore batch requests to achieve the best performance.

TABLE 6
Comparison of technologies for application parallelism

	Applicable to	Integration Overhead
`threading`	Image processor invocation, GCS/Datastore requests	Medium: requires explicit thread creation/starting/joining, requires consciousness of shared data
`multiprocessing.Pool`	Image processor invocation, GCS/Datastore requests	Medium: simple to invoke, requires pickleable parameters, cannot rely on shared data unless we use IPC
Batch requests	Datastore requests	Low, requires coalescing multiple requests into single request

4 ECLIPSE SIMULATOR

The eclipse simulator will be a standalone JavaScript module enabling users to ”preview” the eclipse. It will be designed in a stylized, 2D manner. The simulator will incorporate a time slider to allow users to simulate the eclipse in a time window spanning from 12 hours before the eclipse to 12 hours after it. As users drag the time slider, the eclipse will animate in the simulator window. The view of the eclipse which users are presented will be specific to a location that the user enters. Additionally, the time will be displayed in the simulator based on what location the user enters and the positioning of the time slider.

4.1 User Interface

The user interface for this web based simulator will use 2D animated depictions of the Sun and the Moon as they appear at a user specified time and location. Additionally, the user interface will contain background imagery such as a city or hillside landscape. There will also be a time slider, a location input, and a time display for user's to interact with or view.

For the first possible method, the most basic approach would be to utilize HTML, CSS, and JavaScript for output and input. Altering the locations of the sun and moon would be done by repositioning images of the sun and moon based on location data. While this approach can be simple, repositioning images and making the interface look appealing to the user can be difficult or inefficient.

Another method would be to utilize HTML5's Canvas graphics API. Canvas is capable of rendering graphics directly to the screen via JavaScript. This is otherwise known as ”immediate mode” graphics, which means that the rendered graphics are not saved [25]. Additionally, Canvas draws individual pixels to the screen. For this simulator, the only image objects that we need display differently depending on user input are the Sun and Moon images. Since these are relatively small compared to rest of the simulator, re-rendering the Sun and Moon most likely will not result in poor performance.

The last technology for displaying the simulator would be to use another HTML5 component called Scalar Vector Graphics (SVG). This is a shape based method for describing 2D graphics via XML. SVG utilizes objects to describe images. The browser is capable of re-rendering shapes automatically if attributes to an object are changed [26]. Adding sliders or round buttons to the simulator can also be a straightforward approach with vector graphics.

For the user interface, speed is the most important criteria. All graphics and other simulator resources will need to load in less than 500ms given a 1-10 Mbps internet connection. Additionally the animation used in the simulator should be around 60 frames per second.

Conclusion [See summary of previous discussion in Table 7]: SVG seems to be a better technology to use when creating the user interface because of the interactivity that it has to offer for users. This is due to the fact SVG can process events for each sub-element separately while Canvas must process events for the entire canvas. Additionally, since our simulator requires a few number of objects, we will see better performance benefits than compared to the Canvas or scratch approach.

TABLE 7
Comparison of Possible Technologies for User Interface

	Model	Method	Difficulty of Implementation	Performance
Implement from Scratch	Images	HTML, JavaScript, CSS	May take time to start from scratch	Dependent upon implementation, requires a lot of bandwidth to send large files
Canvas	Pixel Based	JavaScript	Built into HTML5	Smaller surface, larger number of items
SVG	Vector Based	JavaScript, CSS	Built into HTML5	Larger surface, smaller number of items

4.2 Gathering User Latitude/Longitude Information

For the eclipse simulator to be as accurate as possible, we must gather user submitted location data for our computations. There are several options to consider when implementing this. The first of which is having the user only be allowed to enter a specific latitude and longitude when entering their location. While this would be easy to do, it provides a negative user interaction since people generally do not know the specific latitude and longitude of places they will be located for the solar eclipse.

The second option would be to utilize the Google Maps API to display a map [27]. The user would then place a pin at the location that they want the simulator to be based off of. This would require a combination of HTML, CSS, and JavaScript. Displaying a map for the user to interact act with is a much better form of input than the first method. Users don't need to know the latitude and longitude of certain locations, they just need to recognize them via a map. Implementing this would be done by adding a map to the simulator, and then adding a click event function to save the latitude and longitude for when a user clicks a location on the map.

Lastly, we could utilize the Geocoding API which is built into the Google Maps API for converting addresses into geographic coordinates [28]. The user could enter a wide range of input including specific addresses, roads, zip codes, or other options. This data would then be geocoded and will result in the latitude and longitude.

Conclusion [See summary of previous discussion in Table 8]: Overall, reverse geocoding user entered addresses will provide the best functionality. The Google Maps Geocoding API can easily convert these addresses to latitude and longitude coordinates for computing the Sun and Moon's location. While dropping a pin on a map is simple, it requires another large space for user's to interact with and loading an entire map may degrade performance.

TABLE 8
Comparison of Possible Technologies for Lat/Long Information

	Provides Needed Functionality?	User Friendly?	Ease of Setup	API
User Inputs Lat/Long	Yes	No	Very Easy	None
Drop Pin on a Map	Yes	Yes	Medium	Google Maps API
Geocoding	Yes	Yes	Medium	Google Maps Geocoding API

4.3 Displaying Sun/Moon Based on Latitude, Longitude, and Time Information

One of the trickier aspects of the eclipse simulator is altering the positions of the Sun and the Moon based on user entered location and time. The best metric for observing locations of celestial objects is altitude and azimuth. The altitude of an object is the distance it appears above the horizon, while the azimuth is angular distance along the horizon to the object [29]. It is also possible for the Sun and the Moon to be different sizes. We may need to alter the size of the Moon based on location and time while using the simulator. The first method would be to write a JavaScript implementation from scratch for computing the positions of the Sun and Moon based on location and time. This could take significant time to ensure that all equations are correct as well as confirming accurate results.

Another method would utilize a JavaScript library called SunCalc. This can be used to compute sun position, sunlight phases, moon position, and lunar phases based off location and time [30]. This backend library is based off of an article about positions of the Sun as seen from a planet [31]. The documentation is also well written, which makes it very easy to use.

Another JavaScript library exists called Ephemeris [32]. This is very similar to the previous method except that this can be used for ephemeris calculations for a wider range of celestial objects. Additionally, the documentation for this library is very lackluster compared to the SunCalc library.

During the work for the simulator, we noticed that Ephemeris was providing some inaccurate results in our computation. We stumbled upon another JavaScript library called MeeusJs [33]. Although it was not documented too well, it provied much more accurate computations for sun/moon positioning.

Conclusion [See summary of previous discussion in Table 9]: In conclusion, the MeeusJs library is now the library we are using for our computation. Ephemeris and SunCalc both proved to be too inaccurate for what we needed on the simulator. The simulator is working well after replacing Ephemeris with the MeeusJs library.

TABLE 9
Comparison of Possible Technologies for Sun/Moon Display

	Ease of Implementation	External Libaries?	Integration Difficulty	Accurate?
Backend from Scratch	Very Hard	None	Easy - this would be a tailored solution	Possibly
SunCalc Library	Easy	SunCalc	Easy	No
Ephemeris Library	Easy	Ephemeris	Easy	No
MeeusJs Library	Easy	MeeusJs	Easy	Yes

REFERENCES

[1] "Opencv," opencv.org, accessed: 2016-12-13.

[2] "Vxl homepage," vxl.sourceforge.net, accessed: 2016-12-13.

[3] "Simplecv," simplecv.org, accessed: 2016-12-13.

[4] "Stack overflow," stackoverflow.com/tags/opencv, accessed: 2016-12-13.

[5] "What is docker," www.docker.com/what-docker, accessed: 2016-12-13.

[6] "Google cloud platform," cloud.google.com, accessed: 2016-12-13.

[7] "The hough transform," homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT6/node3.html, accessed: 2016-13-13.

[8] K. C. et al., "An efficient circle detection scheme in digital images using ant system algorithm."

[9] S. Mallick, "Blob detection using opencv (python, c++)," www.learnopencv.com/blob-detection-using-opencv-python-c/, accessed: 2016-13-13.

[10] "Simpleblobdetector class reference," /docs.opencv.org/trunk/d0/d7a/classcv_1_1SimpleBlobDetector.html, accessed: 2016-13-13.

[11] K. Larisza and S. McIntosh, "The standardisation and sequencing of solar eclipse images for the eclipse megamovie project."

[12] "Hough circle transform," docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html, accessed: 2016-13-13.

[13] "gsutil tool," cloud.google.com/storage/docs/gsutil, accessed: 2016-10-12.

[14] "Cloud storage," cloud.google.com/storage/, accessed: 2016-10-12.

[15] "Google cloud storage json api overview," cloud.google.com/storage/docs/json_api/, accessed: 2016-10-12.

[16] "Cloud storage client libraries," cloud.google.com/storage/docs/reference/libraries, accessed: 2016-10-12.

[17] "Cloud datastore," cloud.google.com/datastore/, accessed: 2016-10-12.

[18] "Google cloud datastore documentation," cloud.google.com/datastore/docs/, accessed: 2016-10-12.

[19] "Cloud bigtable," cloud.google.com/bigtable/, accessed: 2016-10-12.

[20] "Cloud bigtable documentation," cloud.google.com/bigtable/docs/, accessed: 2016-10-12.

[21] "Connecting to mysql using connector/python," dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html, accessed: 2016-10-12.

[22] J. Noller, "Python threads and the global interpreter lock," jessenoller.com/blog/2009/02/01/python-threads-and-the-global-interpreter-lock, accessed: 2016-10-12.

[23] "multiprocessing - process-based "threading" interface," docs.python.org/2/library/multiprocessing.html, accessed: 2016-10-12.

[24] "Batches," googlecloudplatform.github.io/google-cloud-python/stable/datastore-batches.html, accessed: 2016-10-12.

[25] "Svg vs canvas: how to choose," msdn.microsoft.com/en-us/library/gg193983(v=vs.85).aspx, accessed: 2016-11-09.

[26] "Html5 svg," www.w3schools.com/html/html5_svg.asp, accessed: 2016-11-09.

[27] "Google maps javascript api," developers.google.com/maps/documentation/javascript/3.exp/reference, accessed: 2016-11-09.

[28] "Google maps geocoding api," developers.google.com/maps/documentation/geocoding/intro, accessed: 2016-11-09.

[29] D. M. H. C. University, "Altitude and azimut," http://www.astro.cornell.edu/academics/courses/astro201/alt_az.htm, accessed: 2016-11-09.

[30] V. Agafonkin, "Suncalc," github.com/mourner/suncalc, accessed: 2016-11-09.

[31] "Astronomy answers position of the sun," http://aa.quae.nl/en/reken/zonpositie.html, accessed: 2016-11-09.

[32] Mikhail, "Ephemeris," github.com/mivion/ephemeris, accessed: 2016-11-09.

[33] F. Soldati, "Meeusjs," https://github.com/Fabiz/MeeusJs, accessed: 2017-1-09.