A Web-Based Visualization Framework in Support of Crisis Informatics Research

A thesis submitted in partial satisfaction of the requirements for the degree Bachelors of Science in Computer Science

Fall 2011

Jeff Taggart
Department of Computer Science
University of Colorado, Boulder

Advisor:

Kenneth Anderson, PhD
Department of Computer Science
University of Colorado, Boulder
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2. The Problem</td>
<td>5</td>
</tr>
<tr>
<td>3. The Solution</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Visualization Management</td>
<td>7</td>
</tr>
<tr>
<td>3.1.1 Visualization Management Example</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Object Management</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 Groups Example</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Task Scheduling</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Scheduling Example</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Event Handling</td>
<td>18</td>
</tr>
<tr>
<td>3.4.1 Events Example</td>
<td>19</td>
</tr>
<tr>
<td>4. Framework</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Coding Conventions</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Framework Design</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Engine</td>
<td>23</td>
</tr>
<tr>
<td>4.3.1 VisualizationManager</td>
<td>23</td>
</tr>
<tr>
<td>4.3.2 Visualization</td>
<td>24</td>
</tr>
<tr>
<td>4.3.3 Container</td>
<td>26</td>
</tr>
<tr>
<td>4.3.4 Task Scheduler</td>
<td>27</td>
</tr>
<tr>
<td>4.3.5 Event Model</td>
<td>30</td>
</tr>
<tr>
<td>4.3.5 Elements</td>
<td>31</td>
</tr>
<tr>
<td>4.3.4 Runtime</td>
<td>33</td>
</tr>
<tr>
<td>4.3.5 Renderer</td>
<td>34</td>
</tr>
<tr>
<td>4.4 API</td>
<td>34</td>
</tr>
<tr>
<td>4.4.1 EPIC API</td>
<td>35</td>
</tr>
<tr>
<td>4.5 Utility</td>
<td>35</td>
</tr>
<tr>
<td>4.5.1 Namespace</td>
<td>36</td>
</tr>
</tbody>
</table>
1. Introduction

Since its inception, Twitter has been used across the world for information-sharing purposes. Its micro-blogging platform enables users to quickly share snippets of information regarding any event. It is used by millions of users every day to quickly and efficiently share information. Currently, Twitter is ranked number nine on the world's most popular sites [1]. It represents one of the largest—if not the largest—social dataset in existence. While much of Twitter is filled with uneventful, personal information-sharing, there is a trend towards users using it for more practical purposes.

Recently, the growing field of crisis informatics, which is concerned with how social media and technology affects crisis and disaster response, has shown that Twitter is being adopted for more serious information-sharing. One study, conducted across four "high profile, mass convergence event" suggests that Twitter use during disaster showed increased information sharing and trading [2]. Another study observed "digital volunteers," in the aftermath of the Haiti disaster and described their self-organization techniques during times of crisis [3]. By using a special micro syntax—through the use of hash tags—users were able to increase the information capacity of their tweets. These altered, more useful tweets have important implications on disaster response.

Currently, mainstream disaster visualizations of social media focus on illustrating the increased traffic of social media sites spatially during crisis. An infographic about the Chilean mining disaster illustrates the “explosion” of twitter use during emergency [4]. Another site built an infographic that illustrates that the heaviest use of Twitter during the Queensland Floods occurred on Australia's East Coast, as expected [5]. The site observed Twitter use during the Queensland Floods, pulling data with hash tags and noted that the general feel of the tweets was positive. This indicates that people are interested in helping during a disaster and social platforms enable them to do so.

Liu et al., in a survey of current crisis web mash-ups, notes that most web-based mash-ups surrounding disaster have strong spatial and temporal aspects to them. In times of disaster, people need to know when and where events occur. Liu et al. states that these new uses of web mash-ups have the “potential to possibly challenge or extend these space-and-time models" Liu et al. go on to make suggestions for future design directions of next generation crisis support tools [5]. One interesting suggestion is integrating real-time updates with the ability to travel temporally through a disaster. This would enable users to gain an understanding of real-time developments and keep up with the progression of a crisis. The authors note that using spatiotemporal data from past crises has potential in informing current or future emergencies.

Currently there are several visualization libraries that are available to crisis informatics researchers for displaying social media data collected during an emergency. Unfortunately, these frameworks lack any capabilities outside of basic display functionality. In order to design and develop the next-generation of crisis visualizations and mash-ups, a more robust toolset is needed. In this paper, I present the architecture and prototype implementation of such a framework, which attempts to meet the needs of Project EPIC, a large research project investigating many different aspects of the field of crisis informatics [6].
2. The Problem

Current visualization libraries provide useful features, but due to the limited scope of their design, they do not easily meet the needs of crisis informatics researchers. Many libraries have great charting capabilities, for instance, but they are more oriented around abstract mathematical ideas. For social media data, this is a problem. For example, if you want to plot the amount of tweets per person, or tweets per region, or another mathematically based idea, these libraries are more than capable of helping the developer plot those concepts.

The problem, however, is that Twitter data does not nicely map to these constructs. Crisis informatics researchers are thus faced with a complex, tightly-interrelated system that does not easily translate into the abstract mathematical concepts around which the existing frameworks are designed. How should a crisis informatics researcher represent interactions between Twitter users? How should the clustering of tweets by topic be represented or how should multiple tweet streams be displayed on a timeline in real time using these visualization libraries? Thus, the biggest challenge in using these visualization frameworks is that crisis informatics data does not fit nicely into their toolsets and APIs.

For example, Raphael is a great library for charting [7]. The problem with charting is that you cannot represent the abstract ideas found in social media data well. This causes a researcher to move to a more powerful, but more complex library such as Processing [8]. Processing is nice because you can implement any abstract idea you wish. It is a complete visualization library in the most basic sense. However, with that power comes a loss of the domain-specific functionality (i.e. charting) that you get from more specialized libraries.

With a library such as d3.js [9], if you fit your data to their specific visualization techniques, it is wonderfully easy to produce a very impressive, informative visualization. However, you quickly find yourself at a loss if you try to expand beyond those bounds. This is unfortunate, because the point of these libraries is to produce useful visualization quickly, without requiring vast knowledge of graphics programming. Many researchers that want to visualize ideas are not software engineers. Libraries that make visualization easy are wonderful. And these libraries are great, if it fits your problem. The problem with the datasets being produced by Project EPIC is that they do not easily fit the expectations or assumptions of many of the available visualization libraries. You cannot simply chart social media data.

Another issue is the size and complexity of the potential visualizations, tools, and mash-ups required to address the concerns of crisis management. Most visualization libraries are designed around quick, easy visualizations. Once you try to build a larger system, the libraries provide little functionality in managing user interaction, events, and objects. With d3.js, you get some beautiful visualization capabilities. When you look at the examples provided, however, you realize its intended use is 30-50 line, small programs that visualize data with the provided visualization techniques. A user runs into issues when they try to expand the system into a more complex visualization, especially if they need to modify one of the pre-packaged visualizations.

Using d3.js, I attempted to design a visualization representing Twitter data using a circle packing technique [10]. I chose d3.js because it makes use of the SVG library built into modern web browsers. This makes it straightforward to use JavaScript to bind event listeners on elements of the visualization and thus enable user interaction with the visualization. Circle packing is a visualization
technique designed to represent hierarchical data, where no overlapping occurs and all circles touch each other. This is a useful technique to understand tree-structured data. While you can certainly represent Twitter data as hierarchical, this did not fit my specific use case. The goal was to represent Lucene queries as packed circles and illustrate the connections between distinct queries. d3.js offers a very simple API to produce a circle packing visualization, however I quickly reached the bounds of the usefulness of this visualization technique. I wanted to expand the visualization to incorporate a Venn diagram idea to show the union of two specific Lucene searches. For example, if you had searched "dog AND lost" and "cat AND lost" and wanted to see the common tweets between the two, it would be nice to represent the intersection using typical Venn diagram conventions. With d3.js, short of modifying the library code itself, a Venn diagram visualization is simply not possible. This is not ideal.

To solve this problem, one could turn to a more powerful library, such as Processing.js. However, you lose a lot of nice functionality that you get for free with d3.js, but conversely are more free to express exactly what you need. The problem I quickly encountered was that I lost all of the eventful and interactive capability of SVG, as Processing.js makes use of the canvas element of HTML5. This brought up a smattering of other questions. How would I integrate Processing into Project EPIC’s data collection infrastructure? Would I be able to communicate directly with Twitter from within the code managing the visualization? What about talking to X, where X is a hot new social media site? What about object management? What about scheduling periodic tasks? With these libraries, you must write all of this functionality yourself. Indeed, this is functionality that will be needed in the creation of non-trivial visualizations intended to support the analysis of large-scale data sets, such as the ones being collected by Project EPIC.

One can imagine when you create a new visualization, you have common things that exist across all visualizations. For example, you create a visualization in Processing.js, based on some sort of node visualization. If you want to click on a node, this interaction happens differently across frameworks. A framework such as Processing.js forces you to write the logic yourself, whereas d3.js comes with that functionality baked in. Another interesting problem is sending an event to all nodes at once. All of these visualization libraries only give you the drawing aspects combined with library specific capabilities and SVG functionality if the library decided to use SVG.

These problems led me to the conclusion that I needed to design a visualization framework that could more directly address the needs of Project EPIC, as well other researchers working in the crisis informatics domain.
3. The Solution

I have attempted to design and implement a framework that will make the creation of visualizations, mash-ups, and tools easier. This framework attempts to provide the common functionality that I found useful when attempting to implement various visualization techniques. The key parts I felt that were missing from other frameworks are the ability to manage multiple visualizations and objects, task scheduling, and event management (that is, support for user interaction).

At a high level, the framework allows developers the ability to conceptualize and organize their ideas. If developers utilize the objects provided by this framework, they do not have to worry about building their code in a clean and separated way. The framework provides various objects for managing this organization, allowing developers to focus on the design of their visualizations.

This is useful in the crisis informatics space because it meets the unique needs of crisis visualizations and mash-ups. To build these necessary tools, developers and researchers need to use various third party libraries, which can become complex to manage, and on top of this onerous task is the complexity of the visualization libraries themselves. My framework attempts to alleviate these difficulties by taking on many of the organizational tasks, hiding some of the complexities of the underlying toolkits and providing clear, straightforward paths to creating visualizations, handling events and making it easy to schedule periodic tasks.

3.1 Visualization Management

The ability to manage multiple visualizations is key. Each visualization library assumes you will only ever use their framework to complete your tasks. This is a common problem with assembling software systems from multiple libraries, each library assumes it is in control [11].

My framework is designed to provide the ability to quickly switch between multiple visualizations, all with different renderers that utilize different underlying libraries, thus enabling developers to gain productivity from all available third party libraries. This is a very powerful concept. It allows the developer to generate a common user interface while taking advantage of multiple visualizations engines.

For example, if you had two distinct visualizations that both used the same service to gather data, utilizing my framework makes this goal simple. A developer creates each visualization and registers them with a visualization manager. The user interface widgets that control the visualizations also should also be registered with the visualization manager. Now, a developer can build the use the common pieces of the engine and user interface with both visualizations. The manager can start, stop, and switch out visualizations as needed. Fig. 3A shows what these relationships look like.
This diagram shows how the visualization manager sits in between the global user interface and specific visualizations. The green visualization represents an active visualization. An active visualization is typically any visualization that is rendering visual elements to the screen. The red visualization represents a visualization that is not active. More than one visualization can be active at the same time.

Within this framework, the rendering and runtime pieces of the engine are left mostly up to the developer. Since this is the case, a developer is free to use any libraries (or none at all) to build each specific visualization. A developer, working within my framework, uses the visualization manager to build a higher level tool. The tool or mash-up becomes agnostic to the underlying visualizations. This allows the developer to think at a higher level and plug in libraries and functionality as necessary. The following section walks through an example illustrating this concept.

### 3.1.1 Visualization Management Example

The following example is a tool that utilizes two different rendering engines for visualizing a data set in two unique ways. The libraries used are d3.js and Processing.js. This tool uses Project EPIC’s data collection and analytics infrastructure to access Twitter data [12]. The tool runs a Lucene [13] search against a specific disaster event, or an aggregation of events. The purpose of this example is not to comment on useful visualization techniques, but to show how my framework enables developers to create complex tools. Fig 3B. shows the user interface for the tool. Initially the tool is rather uninteresting, as no data has been loaded.
The dropdown menu on the left represents all events that Project EPIC’s search API knows about. Each event is a distinct dataset. The “all” option searches against every dataset. To start visualizing data, the end-user types a query and clicks the “Search” button.

The “Toggle Vis” button will cause the current visualization to be swapped out. Once data has been loaded, i.e. searching, the user does not have to search again. The data will be automatically loaded into each visualization when a search is performed.

The first visualization, which shows tweets represented on a timeline, uses Processing.js to render a timeline of tweets within a group. This visualization is active by default.

Each group represents one search. The tweets are distributed by time across the line. The oldest tweet is to the left, and the newest is on the right. Fig. 3C shows the search ‘help AND lost’ followed by ‘dog AND found’ against all databases.
After searching ‘help AND lost’ followed by ‘dog AND found’ the visualization renders a tweet group. Each blue dot surrounded with a white border represents a single tweet. The client side search service only retrieves the top fifty tweets for a given search term.

When a search occurs, the search button invokes two callback function, each registered with a separate visualization. This causes the data to load into each visualization. Only one is shown at a time. Fig. 3D shows the tool after the ‘Toggle Vis’ button has been clicked.

The circle-pack visualization was created using d3.js. It shows one search result packed into a circle.

In this example, when the ‘Toggle Vis’ button is invoked, the visualization manager tells the timeline visualization to stop itself. This effectively pauses the visualization. After stopping the timeline visualization, the manager starts the circle-pack visualization. The end-user is free to toggle between the two visualizations at any time.

To build a tool like this, the developer has to follow the object creation and registration patterns within this framework. The developer creates a visualization object, subclassing the necessary framework objects to build each visualization. After the visualization has been built, the developer registers the visualization with the master visualization manager. This object controls the life cycle of each visualization. The following code snippet shows how easy it is to register a visualization with the manager.

```javascript
var manager = epic.engineVisualizationManager();
manager.registerVisualization(epic.engineVisualization(...));
```

The specifics of how to create visualizations will be discussed in Section 4.3.1. Once a visualization has been registered in this manner, the developer only needs to call `start` and `stop` on the manager to manage visualization life cycles.

This example shows how useful it is to allow the developer to use more than one visualization. Perhaps a developer has generated several visualizations and wished to bring them together. This framework enables the developer to do this in a straightforward manner.

Fig 3D – The tool after ‘Toggle Vis’ has been clicked. The circle pack visualization is now representing the search ‘help AND lost’
### 3.2 Object Management

Any non-trivial visualization or mash-up will need object management. When a developer wants to display ideas, the natural way to conceptualize them is through objects. For example, if you were describing people visually, a developer may choose to represent them as circles. If there are other, non-human objects to represent, a developer might choose to represent those as squares. Once you get more than a few “groups” of objects, it becomes important to organize these in some way.

Within my framework is a container object that aggregates various visual elements. Registering elements with the container enables this idea of object management in practice. Fig. 3E shows conceptually what this looks like.

![Diagram](image)

**Fig. 3E**: The relationship between a visualization’s container and its visual elements. Each element is stored in specific collections as well as a comprehensive “all” collection.

This diagram shows how a container holds on to all elements. Each time an element is registered, the developer must choose which collection to place the element within. If the developer chooses ‘Collection 1’ then the element will be contained within that collection. If no collection exists, then one is created dynamically. The container also has a collection that represents all of the elements. No matter which collection the developer chooses, the element will end up in the ‘all’ collection as well. A developer can choose to access elements within the container through the ‘all’ collection.

Using this containment idea, the framework allows a developer to conceptualize and group distinct elements. Each visualization has a container where they can register elements. By giving each element group a name, the developer can interact with various sets of elements as needed. Imagine the ability to hide all non-human elements was required. Without proper object management, this task would be tedious and slow. In the previous example, a developer would have to ask each distinct element if they were non-human.

Instead, using this framework, they can organize the elements within a visualization using the provided container. This enables powerful interaction with groups, allowing a developer to ask for
In the previous example, hiding non-human elements becomes trivial. The following section shows a working example of this powerful idea.

### 3.2.1 Groups Example

The following figures represent how a developer might use the framework to organize elements. Let circles be “humans” and squares be “non-human”. Fig. 3F shows how two distinct lines of shapes, one representing the “human” circles and one representing the “non-human” squares.

Fig. 3F: A visualization with multiple groups of elements registered with its container.

The container is represented by the grey square. Each square and circle has been registered with the container. Notice the buttons along the top of the visualization. Each button interacts with a group of elements. An end-user that selects either the “Toggle Circle Group” or “Toggle Square Group” button will cause the visualization to hide one group, respectively. The “Toggle Both” will cause both groups to switch to the opposite state they are currently in, either hidden or visible.

Fig. 3G shows the two states after pressing each button separately.

Fig. 3G: The results of clicking on the first two “toggle” buttons shown in Fig. 3F.

A naïve way to accomplish this effect is to loop through every object and ask if they are a “human” or a “non-human”. However, this is slow and tedious. Using my framework a developer can logically group objects in whatever manner suites their specific needs. Using a container object, each object can be registered. The container manages which object exists in what group. The following code shows how simple it is to register an element with the container.
This code snippet loops through and creates five circles and squares. The developer is calling `registerElement` on the container with a collection argument. By stating a circle goes into ‘circles’, the container places each circle into the ‘circles’ collection. The square objects work in the same manner. When an end-user clicks on a button, the code to switch the visibility of an object is straightforward. The proceeding code snippet shows the code registered with a button to accomplish this goal:

```javascript
var i, circle, square;
var container = this.container;
for (i = 0; i < 5; ++i) {
    circle = Circle();
    container.registerElement('circles', circle);

    square = Square();
    container.registerElement('squares', square);
}
```

`$('#toggle-circles').button().click(function () {
    $.each(container.elements['circles'], function (i, circle) {
        circle.visible = !circle.visible;
    });
});`

A developer simply has to ask the container for the elements that are registered under the collection ‘circles’. This will return all elements registered under that name.

One convenient operation the container performs upon object registration is to register all elements under an ‘all’ collection. This allows for a developer to grab every element if needed. To illustrate this, the previous code becomes:

```javascript
$('#toggle-circles').button().click(function () {
    $.each(container.elements['all'], function (i, element) {
        element.visible = !element.visible;
    });
});
```

Notice the second line where ‘circle’ changed to ‘all’. Remember that each element that is registered with the container is placed into the ‘all’ collection. This small change is all that’s needed to grab every element that’s been registered with the container. This functionality is particularly useful if an entire visualization must be hidden.

### 3.3 Task Scheduling

Task scheduling functionality makes it easy to quickly add tasks, with fine-grained control to modify
their associated visualizations. The scheduler in my framework manages both long-running and one-time tasks. This enables the developer to create tasks that have a lifecycle, performing a task every so often, or a task that performs some task at some point in the future. To use the task scheduler functionality, a developer registers a task scheduler object with a visualization. After the object is registered, the developer simply needs to call the tick function from within their runtime’s main loop. The tick function will perform any task that is determined to be performed at that time.

Registering a task is easy. A developer simply has to create a task object and add it to the task scheduler. The task can be in a paused state as well, providing the ability to pause tasks temporarily. Fig. 3H shows conceptually how the runtime, task scheduler, and tasks interact.

![Fig. 3H: Task scheduling concepts in the visualization framework.](image)

The runtime, which is defined by the developer, calls tick on the scheduler, which in turn calls tick on each one of its tasks. The tasks in green represent tasks that are active. The red tasks are paused. From the runtime’s perspective, it does not care how many tasks have been registered, if any at all. It simply tells the scheduler each time there has been an update.

A developer may register a task from anywhere in his code. The only requirement is to create a task object and add it to the scheduler. All that’s required for this to work is a runtime loop that calls tick on the scheduler. This is powerful, as it enables functionality to be tied into the main runtime loop dynamically.

Tasks are best used to manage animations. If a developer needs to perform an animation, they can create a task and register it. After that, the task scheduler takes care of the rest. It will perform the task until the task informs the scheduler it is complete. If the developer decides to pause the task, they simply tell the scheduler to do so. This fine-grained control over tasks is enabled by the object management philosophy described in the previous section. Each task is registered with a unique name and can be organized into specific groups. This is particularly useful in pausing and restoring tasks. A developer can choose to pause one task or a group of tasks. The following section shows how the task scheduler is used.
3.3.1 Scheduling Example

As previously mentioned, a great use of the task scheduler is for animation. Most visualizations require some sort of animation to reinforce a certain point. One way to accomplish animation is writing the animation code either in the main runtime loop, or having an object know about its specific animations. Using the task scheduler properly enables a much more powerful way to accomplish this same functionality.

In this example, we have a row of circles. The visualization has three requirements; each circle must pulse—change color every few seconds—and move to the top of the container when told to do so. An end-user also needs the ability to toggle each animation on and off. Fig. 3I shows the user interface for this example.

An end-user can cause the circles to move to the top of the visualization by clicking the 'Move Circles Up' button. The delay indicates how often the circles will move and the total moves is how many moves will be performed. Providing different values for these two inputs causes varying animations to occur; fast or slow and choppy or smooth.

The 'Toggle Pulse' and 'Toggle Color Change' buttons control the pulse and color change animations. An end-user can toggle each specific animation on and off using these buttons.

The 'Toggle Daemons' button causes both animations to be paused or restored depending on its current state. For example, if the pulse animation was on and the color change animation was off, the pulse would be turned off and the color change would be turned on. The task scheduler’s tick function is called every time the runtime loop is executed. Fig. 3J shows the transition after a few ticks have passed.

Each circle color has changed from orange to teal. The stroke width has also increased. The color animation is abrupt while the stroke width is a gradual transition. This serves to illustrate the delay functionality with a task. The delay can be used to simulate smooth transitions along side occasional, instantaneous changes.

Fig. 3I: The user interface of the task scheduling example.
To accomplish the abrupt color change, a task is registered with a relatively long delay. In this specific example, the delay is set to 50. Each tick of the task scheduler causes the tick count for the specific task to increase. When 50 ticks have passed, the task scheduler invokes the task. This causes an abrupt color change. The task then lies dormant until another 50 ticks have passed. This code snippet shows how to create this specific task.

```javascript
var colorTask = epic.engine.Task({
   name: "color-pulse",
   type: "daemon",
   exec: function () {
      var circles = container.elements['circles'];
      $.each(circles, function (i, circle) {
         if (circle.color === red) {
            circle.color = teal;
         } else {
            circle.color = red;
         }
      });
      return true;
   },
   delay: 50
});
```

The specifics of task creation will be discussed in Section 4.3.4. The important piece to note is the delay. If a developer wanted to increase how often the color change occurred, they would simply decrease the delay count.

The pulse animation is almost identical, but it has a much lower delay. Using a lower delay, we can simulate a circle pulsing but increasing and decreasing the stoke width gradually. The following code shows how to create the pulse effect.

![Fig. 3](image.jpg): The task scheduling example after a few frames of animation have passed.
Notice how both tasks have been registered with the name ‘daemon’. This declares that both tasks are of that type. This allows the developer to stop and start the tasks by type. By pressing the ‘Toggle Daemon’ button causes the scheduler to look up tasks by that type name and pause them. Similar to types is the name attribute. This is used to identify a task, which can also be used to start and stop a task.

The third animation included in this example is moving the circles to the top of the container over time. Fig. 3K shows the circles after the ‘Move Circles Up’ button has been pressed. The task that handles this animation is a bit more complex than the previous two. When an end-user clicks the ‘Move Circles Up’ button, a task is created dynamically. This task contains the delay that is specified by the end-user. Each time the task is executed it moves the circles up by a small amount. That amount is determined by how many steps the end-user indicates. When the circles have moved to the top, the task indicates to the scheduler that it is completed and the

```javascript
var width = 0;
var up = true;
var pulseTask = epic.engine.Task({
    name: "stroke-pulse",
    type: "daemon",
    exec: function () {
        if (width > 10) up = false;
        if (width < 2) up = true;
        if (up) width += 0.5; else width -= 0.5;
        $.each(container.elements['circles'],
            function (i,circle) {
                circle.strokeWidth = width;
            });
        circle.strokeWidth = width;
        return true;
    },
    delay: 5
});
```

Fig. 3K: The circles in the task scheduling example after being told to move up.
scheduler removes the task.

These examples of the task scheduler show how powerful tying functionality into the main draw loop can be. By allowing the developer to create tasks with various delays and life cycles, it enables anything from a simple one-off tasks to long-running periodic tasks.

### 3.4 Event Handling

Event handling is another core piece of the framework. The ability to register and trigger events allows the developer to keep code modularized. This functionality in modern web browsers is built into the DOM (Document Object Model) and SVG but is missing in the HTML5 canvas element. My framework brings this functionality back to the canvas. This allows for event management across all three of these browser elements and enables the creation of visualizations that behave consistently regardless of which of these features the visualization uses.

An example of how a developer could make use of this piece of the framework is with user interfaces. A common pattern in JavaScript to create interactive interfaces is to register callback functions with elements on the screen. A developer can use the callback functions to trigger code that then manipulates the visual elements via changes to the DOM. My framework enables the same sort of pattern with the HTML5 canvas element.

For example, a developer can create two button elements that both trigger events on the container. Fig. 3L shows how a button triggers an event on the container object, which in turn triggers the same event on all registered elements.

![Diagram showing event handling](image)

**Fig. 3L:** Enabling user interactions via events within the framework.

In this example, the container has several eventful objects registered. The green eventful objects indicate that the object has an event handler for the given event. The red indicates that the object does not. When the button triggers an event on the container, it in turn delegates (triggers) the event on each one of the objects. Because of the way the framework was designed, if an object does not
have an event handler for the specific event, it simply ignores it. This creates a way for the developer
to not have to care about the specifics of the visualization at every point in his code.

The buttons in this example do not know what the container contains. They are completely separate
from the implementation of the visualization. This allows objects to be created and have event
handlers registered for them dynamically. The only connection between the button and the objects is
the container. From the button’s standpoint, it is perfectly fine if no objects respond to its event.

Since an eventful object ignores any event it does not care about, the container is also free to ignore
the details about which element is interested in which event. It simply sends each event to all of its
objects, and the ones that care will respond. The next section illustrates this simple concept.

### 3.4.1 Events Example

This example provides the ability for an end-user to change the all circles to a different color
simultaneously. Fig. 3M shows the interface for this example.

![Events Example](image)

**Fig. 3M:** The user interface for the event management example.

The end-user can click on ‘Change to Teal’ to change each circle from orange to teal. Conversely, clicking ‘Change to Red’ will cause the circles to turn red.

Fig. 3N shows how the state of the circles changes from red to teal upon clicking the appropriate button.

![User Interface](image)

**Fig. 3N:** The user interface responding to the events generated by its buttons.

When an end-user clicks a button, the button triggers a change color event on the container. The
container, in turn, triggers the same event on each one of the five circles. This code shows how the
button triggers the appropriate event.

```javascript
$('#change_teal').button()
    .click(function () {
        container.trigger('teal');
    });
```
This snippet shows how simple it is to trigger an event. The user interface has absolutely no idea about the objects that the visualization contains. In the same fashion, the following code shows how the container delegates the event to each registered object.

```javascript
container.on('teal', function () {
    $.each(this.elements['all'], function (i, element) {
        element.trigger('teal');
    });
});
```

As can be seen above, the container doesn’t care about the specific objects registered with it at the time of the event. It simply triggers the event on each of its elements. This pattern provides a very clean separation of concerns. By having the object handle the event itself, no other objects have to care about whether or not it responds to a message. There could be a square added to this visualization dynamically, and if it cared to turn teal, it would.
4. Framework

I chose to implement an HTML5 JavaScript toolset that works with both SVG and Canvas, although most of the engine is more pertinent to the latter. The reason that I chose HTML5 and Canvas is that they represent a rapidly growing technology that allows access to visualizations and mash-ups to anyone with a browser. As an added benefit, HTML5 and Canvas support is also being added to the browsers that ship on mobile devices. In a disaster situation, first responders and volunteers cannot necessarily be bothered to install specialized software to gain access to important information about a disaster. It is almost an expectation that useful software for these purposes should be browser accessible.

This framework provides a clean simple way to: handle events across browser-specific technologies such as the DOM, SVG, and Canvas; allowing access to custom API services; handling of UI events; and object management across SVG and Canvas. Since a user of this library will be expecting HTML5, the engine targets ECMAScript 5 as its target implementation of the JavaScript language [14]. A listing of the compatibilities of different popular browsers can be found at Juriy Zaytsev’s ECMAScript 5 compatibility table [15].

The framework also contains several components that were a common need of any visualization or mash-up. This includes event handling for the canvas, task scheduling, layer management, object management, a runtime environment, visualizations management, utility functions, APIs, and services. The goal was to provide a useful foundation for future mash-ups and visualizations to use. You do not get cheap and easy visualizations out of the box with this framework. Instead, this framework provides a foundation with large, complex systems in mind.

As such, the majority of the framework’s functionality resides in its engine. Other top-level components are provided as an example of the types of things that can be built on top of the engine. For instance, the prototype that I constructed contains several examples that make use of specific parts of the framework and one example that illustrates how to create a more complex visualization.

4.1 Coding Conventions

In order to help the reader understand code snippets vs. API documentation, they appear differently throughout. Code that appears anywhere in the document, including inline with normal text, is presented in Inconsolata, a fixed-width font. A code snippet (as seen previously) appears in a grey box like so:

```javascript
var codeSnippet = “This is a code snippet”;
```

API docs appear in this format:

```
Attributes

_privateAttribute: Type
Description of the attribute.
```
attribute: Dictionary<Object>
Description of the attribute.

Methods

_privateMethod(parameter: Type): Void
Description of method.

publicMethod(parameter: Type): Array<Type>
Description of method.

Attributes are object attributes defined by the propertiesObject (see Appendix A). If public, these attributes are used by other objects. An attribute that is prefixed with ‘_’ indicates a private method that should only be accessed from within the class.

The Type flag indicates the required type of the attribute or parameter, i.e. String. A Void return type indicates there is no return type. This type may be Undefined, Null, or garbage and should not be used. One special type is Dictionary. This is simply a generic JavaScript object intended to be used as a dictionary. This means treat it like a key-value store. The key will always be a String.

If an Array or some other collection, such as Dictionary, is being returned the Type contained within the angle brackets, i.e. <Type>, indicates what type of object the collection contains.

Private methods, also indicated with a ‘_’ prefix, are included in the API documentation since the framework is intended for working from within the actual files. This framework is a foundation for a larger, complex application to be built within, and is not intended to be a third party framework that you include. Private methods are potentially useful to know about for a user of the framework. I prefer to over-document then under-document.

Please be aware that, since JavaScript is a dynamic language, there is no type checking. The types presented in this framework must be understood and used properly by a user of the framework. A lack of discipline regarding these types leads to undefined behavior. Please see Appendix B for a discussion of the object-oriented design of the framework.

4.2 Framework Design

The framework is implemented using object-oriented techniques, with each component being distinct from one another. This allows for users to inherit useful functionality from the framework’s classes, or define and plug in their own.

For example, if a user wants to only use the scheduler from the engine portion of the framework, it is as simple as calling epic.engine.Scheduler(), to create a scheduler object. Each component of the framework is implemented such that it can standalone from the other components. They are designed to work in concert with each other but if a specific implementation provided by the framework does not meet a user’s needs, they can substitute in a different implementation as long as it conforms to certain interfaces.

The following sections describe the four major parts of the framework. The main component is the Engine, which provides core functionality to various visualizations. The other parts include: Utility,
library of useful utility routines; Services, which is a section to define services needed by visualizations; and Api, a place to implement the logic needed to talk to various third party applications. Most work was done in the engine, with the other parts of the framework providing working examples of how to separate concerns into logical groupings.

4.3 Engine

The Engine is the core of the framework. The goal was to create a consistent visualization engine that provided much of the functionality missing from traditional visualization libraries, while leaving the actual rendering to specific users of the framework.

The engine remains decoupled from the specific rendering library, and defers to the user to implement graphics specific functionality with their library of choice. This has the advantage of giving the end user the freedom to render their visualization how they want, while getting useful functionality that is not specific to rendering. See Appendix A for the Engine’s class diagram.

A great example of this is the event model baked into the engine. Since HTML5’s Canvas element does not support events like SVG or DOM elements, it forces users to implement their own event management. By taking an object-oriented approach in the engine, adding event management is simple. The Engine provides users with a consistent model for events. By separating this from the renderer, a user can trigger events in the same manner across visualizations and user interfaces.

4.3.1 VisualizationManager

The VisualizationManager contains all registered visualizations. This is the master object that knows about all visualizations and global user interface elements within a page. A user must register all visualizations and global UI elements with the VisualizationManager. User interface elements can be managed outside of specific containers. This enables users to build their UI separate from specific visualizations. The VisualizationManager can handle events and delegate to UI elements as needed. If needed, one can have visualization specific UI elements managed by a container.

This object manages the lifecycle of each visualization. Using the start, stop, and replace methods, a developer can easily activate, replace, or stop visualizations. The replace method is particularly useful if the application has one visualization pane, and wants to easily swap out various visualizations. Since this object is responsible for the lifecycle of everything else that exists on the page, only one of these should exist in an application.

The API for the VisualizationManager is as follows:

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>_visualizations: Dictionary&lt;epic.engine.Visualization&gt;</td>
</tr>
<tr>
<td>This is a private attribute that is an object that holds onto visualizations. The visualizations are of type epic.engine.Visualization. Each visualization is indexed by its name attribute.</td>
</tr>
<tr>
<td>window: epic.engine.Window</td>
</tr>
<tr>
<td>This is a public attribute that represents the width and the height of the browser window. It is a simple object with two attributes, width and height.</td>
</tr>
</tbody>
</table>
uiElements: Array<epic.engine.Ui>
This is a public attribute that holds all global UI elements that can be accessed by a visualization to display data. The elements are of type epic.engine.elements.Ui.

Methods

This function registers a global user interface element.

registerVisualization(visualization: epic.engine.Visualization): Void
Registers a visualization with the VisualizationManager. The visualization parameter must be of type epic.engine.Visualization. The name key on the visualization is how the manager finds a specific visualization.

getActive(): Array<epic.engine.Visualization>
Returns an array of all active visualizations. Any number of visualizations may be active at once. If no visualizations are active, the function returns an empty array.

getVisualization(name: String)
Returns the visualization registered under the specified name. If no visualization is registered by the name, undefined is returned.

start(name: String)
Starts the visualization specified by the name argument. Starting a Visualization involves setting its active flag to true and call start(). If none is registered, the function returns false. If the Visualization is successfully started, the return value is true.

stop(name: String)
Stops the visualization specified by the name argument. The function expects a string. This function will set the active flag to false and call stop() on the visualization. If no visualization is registered, the function returns false. If the visualization is successfully stopped, the return value is true.

replace(oldVis: String, newVis: String)
Stops a visualization, specified by the oldVis argument, and starts a new visualization, specified by the newVis argument. Both arguments are expected to be strings. If the old visualization is successfully stopped and the new visualization is successfully started, the function returns true, otherwise it returns false.

---

4.3.2 Visualization

A Visualization object is the container of the various components of a visualization. The purpose of this object is to receive commands from the VisualizationManager and delegate them to the appropriate object.

To create a Visualization object, an initialization hash table must be passed as the only parameter. This hash table initializes the attributes of a visualization. The only optional attributes are active, services, and apis. The following is an example of creating a Visualization object:
The following is the API for the `Visualization` object:

```javascript
epic.engine.Visualization({
    name: "name of visualization",
    active: boolean,
    container: epic.engine.container(universe),
    scheduler: epic.engine.scheduler(),
    apis: {
        epicApi: epic.api.Epic()
    }

    services: {
        tweetService: epic.service.tweetService()
    },
    renderer: epic.vis.processing.renderer(),
    runtime: epic.vis.processing.runtime()
});
```

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name:</strong> String</td>
<td>The name, of type String, used to refer to a specific visualization. This should be a globally unique name to avoid name collisions within the <code>VisualizationManager</code>. Duplicate names produce undefined behavior.</td>
</tr>
<tr>
<td><strong>active:</strong> Boolean</td>
<td>A Boolean that determines if a visualization is active or not. The <code>VisualizationManager</code> toggles this flag when it starts and stops a visualization. Setting this flag outside of the manager produces undefined behavior.</td>
</tr>
<tr>
<td><strong>container:</strong> epic.engine.Container</td>
<td>The container that holds the elements of the visualization. There is only one container per visualization.</td>
</tr>
<tr>
<td><strong>scheduler:</strong> epic.engine.Scheduler</td>
<td>The task scheduler. There should only be one scheduler per visualization.</td>
</tr>
<tr>
<td><strong>apis:</strong> Object</td>
<td>This object holds any libraries that communicate with third party APIs, such as Project EPIC’s data collection infrastructure or Twitter. Each API should have a unique name that is its index in the <code>apis</code> object.</td>
</tr>
<tr>
<td><strong>services:</strong> Object</td>
<td>This object holds any services needed by the visualization. Typically, these objects are the consumers of API objects. Each service should have a unique name that is its index in the <code>services</code> object.</td>
</tr>
<tr>
<td><strong>renderer:</strong> epic.engine.Renderer</td>
<td>The renderer object that handles the drawing of all elements.</td>
</tr>
<tr>
<td><strong>runtime:</strong> epic.engine.Runtime</td>
<td>The runtime object that handles the orchestration of the visualization.</td>
</tr>
</tbody>
</table>
Methods

\texttt{start()}

Method that is called to start a visualization. This method should be overridden by a subclass to define behavior appropriate to starting a visualization. It does nothing by default.

\texttt{stop()}

Method that is called to stop a visualization. This method should be overridden by a subclass to define behavior appropriate to stopping a visualization. It does nothing by default.

4.3.3 Container

The Container is the object that can be used to manage all elements within a specific visualization. Since the container manages all objects, it understands the dimensions of the visualizations. When a user is interacting with a visualization, they will be clicking on the container first, which will delegate events to the appropriate objects.

This object is a subclass of \texttt{epic.engine.Eventful}. This provides a clean way to trigger events on an entire visualization. One event that the \texttt{epic.engine.Runtime} registers for the document window is the ‘resize’ event. When this happens, the window will trigger an event on its corresponding container. Since each visualization has its own distinct runtime and container, window resize events will occur on all containers regardless of the state of the visualization (active or not).

The \texttt{Container} contains any elements related to a specific visualization. These elements should be of type \texttt{epic.engine.Element}. A user should register all elements with the container. This is useful because a user can trigger an event on the container, such as hide visualization, and it can find the appropriate objects to notify. The hide event on a container is triggered by default by \texttt{runtime.stop()}. A user must register an event listener on the container if they care about the hide event. There is no event handler by default. It is not required, however, since if no event listeners are registered, nothing will happen.

A \texttt{Container} holds on to multiple layers. This allows for fine grain control over which elements receive events. For example, if you wanted to trigger an event on all events, you could simply loop through \texttt{elements} array and trigger an event. But if a user wanted to trigger an event on only elements of one layer, the \texttt{layers} attribute allows the user to do so.

The constructor function expects an \texttt{epic.engineVisualizationManager} as the only parameter. The manager should be the global manager for all visualizations. This is necessary to provide a way to trigger global events from within the visualizations, such as displaying information on the global UI.

The API for the \texttt{Container} is as follows:
Attributes

**manager: epic.engine.VisualizationManager**
This is a handle back to the global VisualizationManager. This allows for the container to trigger global events.

**window: Object**
This is a simple object that has two attributes, `width` and `height`. This object represents the current size of the world for the elements within the container.

**layers: Array**
This is an array that contains all of the layers for the container.

**elements: Object**
This is a simple object that contains different groupings of elements. By default, it contains an `all` attribute. All elements registered with the container get placed in the `all` grouping.

Methods

Registers an element with the Container. The collection is the name in the `elements` object. When a user registers an element, pushes the element onto the layers `elements` array, pushes the element into the appropriate collection on the `elements` object, and pushes the element into `all`.

**clearElements(): Void**
This wipes all elements from the `elements` object.

**getElement(x: Integer, y: Integer [, collection: String]): Array**
This method asks each element if it existsAt the given `(x, y)` coordinate. All found elements are returned to the caller. If a Layer is passed in, it will only ask elements that exist on that layer. Only elements that are real and not hidden are searched for.

**setWindowSize(width: Integer, height: Integer): Void**
This method sets the window size of the container. If the window size is different than the old window size, this triggers a 'windowResize' event on itself.

4.3.4 Task Scheduler

The task scheduler provides the ability to schedule tasks to be executed. Tasks can be executed an unlimited or finite amount of times. A user can also specify the delay between executions per specific task. This component of the engine allows the user to perform useful tasks, such as ping a server for updates.

4.3.4.1 Task

If a user wants to create a task, they must construct an `epic.engine.Task` object. A task object holds a function to be executed each `tick + delay`. The `Task` object subclasses `epic.engine.Eventful`, giving it the ability to register events. This is a useful way to trigger events on any number of tasks. The `Scheduler` uses events to inform the task of what state in its life cycle it's currently in. This functionality will be discussed in further detail in Section 4.1.4.2.
The following is an example of how to construct a Task object:

```javascript
var task = epic.engine.Task({
    name: 'task',
    type: 'daemon',
    exec: function() {},
    delay: 100,
    paused: false
});
```

The Task object takes a name attribute, which identifies which task it is. This name should be unique across all tasks. This attribute must be passed into the constructor. To distinguish tasks, the user specifies a type for the task. In the example, the type is ‘daemon’, indicating this task is a background process that occurs every 100 ticks. The default value for type is ‘misc’, indicating it is a generic task with no special type.

To specify the interval of time between task execution, the user specifies a delay amount. The Scheduler uses this number to determine when to execute the task. The delay is how many ticks to skip. A 100-tick delay means the Scheduler lets 100 ticks go by before executing this task. This attribute is set to 0 by default if no value is passed to the constructor.

The pause attribute is used to determine if a Task is active or not. If this flag is set to false, the Task will never be executed. By default this value will be false. A task should not set this value directly, except as a constructor argument, as this will be managed by the Scheduler.

The exec function is the function to be executed each time the Scheduler determines its time to execute that task. When the function is called, the this variable is set to be the specific task. The return value of the function must be true or false. The exec function must return true if the task needs to occur again. If the Task is complete, it should return false, which will cause the Scheduler to remove the Task.

The epic.engine.Task object has the following API:

---

**Attributes**

- `lastExec: Integer`
  The last tick count the task was executed.

- `ticks: Integer`
  How many ticks have passed for the task.

- `name: String`
  The name of the task. This should be unique across all tasks registered with the Scheduler.

- `type: String`
  The type of the task. This attribute is used by the Scheduler to control the life cycle of tasks grouped by type.
delay: Integer
How many ticks to skip between executions.

paused: Boolean
Determines if the task is active or not. If the task is paused, the `exec` function will not be called. The task’s `tick` attribute will still be incremented.

**Methods**

`exec(): Boolean`
This is the function to be executed every time the task is scheduled. The function must return `true` or `false`. If the task returns false, it is no longer valid, and should be removed.

### 4.3.4.2 Scheduler

The `epic.engine.Scheduler` is the object that manages `epic.engine.Task` objects, described in the previous section. Its purpose is to keep track of how many update ticks have occurred and schedule Tasks accordingly. Each Task’s life cycle is managed by the Scheduler, triggering events to inform the task of its current state.

When a task is first registered, the Scheduler triggers a ‘start’ event on the task. The task can bind an event listener to this event to do initialization prior to the main `exec` function being called. The Task can also ignore this event if no initialization is required. Upon the end of a Task’s life, the Scheduler will trigger an ‘end’ event on the Task. In the same fashion as the ‘start’ event, the Task can listen for this event, and do any necessary cleanup after it has done its work.

The Scheduler determines whether to end the life of a Task by checking the Task’s `exec` function’s return value. The value must be a `Boolean`. If the Task returns `false`, the Task will be ended, and the ‘end’ event will be triggered. The Scheduler then removes the Task.

The developer determines the life cycle of the Scheduler. The `tick` member function increments the tick counter and schedules the appropriate Tasks. It is suggested to call `tick` on the Scheduler within their main Runtime loop. Since the user drives the Scheduler, each tick is an undefined amount of time.

The Scheduler maintains an array of Task objects. Each time `tick` is called, the Scheduler determines which tasks should be executed, depending on how many ticks have passed since the task’s last execution time.

The API for `epic.engine.Scheduler` is as follows:

---

**Attributes**

_ticks: Integer_
Tick counter. This attribute is updated every time `tick()` is called.

_tasks: Array<epic.engine.Task>_
The collection of tasks owned by the Scheduler.
Methods

addTask(task: epic.engine.Task): Void
Adds a task to the scheduler.

tick(): Void
This function increments the tick counter and executes and tasks that are eligible. A task is eligible if its `tick - delay` is less than its `lastExec` count. A task’s `exec` function must return `true` or `false` to indicate if it is done.

getTaskByName(name: String): epic.engine.Task
Gets a task by the specified name. If no task by that name is present, the function returns `null`. If there is more than one task registered under the given name, the first match is returned. It is suggested to have each task registered under a unique name for this reason.

getTasksByType(type: String): Array<epic.engine.Task>
Returns all tasks for the given type. If there are no tasks under that type, an empty array is returned.

startTasksByType(type: String): Void
Starts all tasks by the given type. This function sets the `paused` flag to `true` regardless of its previous state. If no tasks are present, nothing happens, so it is safe to call with no registered tasks of that type.

stopTaskByType(type: String): Void
Stops all tasks by the given type. This function sets the `paused` to `false` regardless of its previous state. If no tasks are present, nothing happens, so it is safe to call with no registered tasks of that type.

4.3.5 Event Model

One major component missing from the HTML5 canvas is JavaScript event management. With a library such as d3.js you get event management for free since it utilizes SVG. If you have a great visualization written in d3.js, but you cannot quite get it to fit your data, you may turn to a visualization library based on HTML5’s Canvas element. Moving the data into this new library, one loses the event model and you then find yourself implementing an event model.

The event model provided by this toolset works identical to the event model that is built into the browser in the DOM. By making an element inherit from `Eventful` you now have an element that can handle events. This is convenient if you are toggling between a visualization using SVG and Canvas, you can conceptualize the interaction in the same way. This functionality is especially useful when creating a GUI. A GUI should be consistent across visualizations, and will need to be able to react to events that are sent to it (for example, telling it to display a tweet's information). By having the event model abstracted from the standard JavaScript DOM event model, you would not have to re-implement the interaction between a visualization and its GUI. You just trigger an event in the same way.

To create an `Eventful` object, simply subclass `epic.engine.Eventful`. The event model has the following API:

on(name: String, callback: Function)
Adds a callback function under the specified name. Multiple callback functions can be registered under the same name. Adding a function n times will result in the function being called n times.
off(name: String [, callback: Function])
Removes the specified callback function for the given name. The callback function must be present to
distinguish it from other functions registered under the same event name. If the callback argument is
undefined, all event listeners will be removed for the specified event name. If you wanted to reduce the
number of function calls, you must remove the callback function, and re-add the number of functions back
with on.

trigger(name: String [, data: Object])
Calls functions registered under the specified event name. This function loops through each registered
function and calls the function passing data argument. this is set to the object that registered the event
listener.

4.3.5 Elements

At the heart of any visualization are the objects that are visually displayed on the screen. These are
known as elements within the framework. At a basic level, these objects are essentially dumb data
holders that help the developer organize and store information about a specific element. These
objects are intended to give the developer a basic building block to create more specific and useful
objects. Each object should be subclassed appropriately.

The elements store various bits of information; mostly pertaining to their current state and the
information they hold. Various properties are set on these objects to determine properties such as
visibility, selection state, and its boundaries.

On top of this is basic functionality to determine if it exists at a given x, y coordinate and to set its
boundaries and position. The existence functionality is very naïve and a developer should override it
if they need more fine-grained control over boundaries.

4.3.5.1 Base

The epic.engine.Base object provides basic functionality that is useful to any type of visual
element. This includes position, dimensionality, visibility and other state. It also provides a handle
back to the container so an object may trigger events. This object is intended to be subclassed by the
developer as it lacks enough functionality to be useful. For example, the draw function must be
overridden to actually display anything. This object subclasses epic.engine.Eventful.

This object also holds on to a model object. Since most useful visualizations, especially in crisis
informatics, are data driven, it is useful to have a consistent place to store that data. This provides a
consistent place to place and access data attributed with elements.

The Base object also provides three basic functions and two prototype functions to provide basic
functionality. To set the bounds of a object, call the setPosition function. A developer can set the
bounds, i.e. width and height, of the object by calling setBounds. Existence at a position is
determined by the existsAt function.

Please keep in mind these functions provide very basic functionality. The setBounds and
existsAt functions are based on a rectangular width and height. This is only ideal for rectangular
objects. These functions exists to aid in prototyping. For more complex visualizations, consider
overriding.
The API for the `Base` object is as follows:

---

### Attributes

- **x**: Integer
  X position of the object.

- **y**: Integer
  Y position of the object.

- **z**: Integer
  Z position of the object.

- **width**: Integer
  Width of the object. This is used to determine boundaries. Typically this is set by the `setBounds` function.

- **height**: Integer
  Height of the object. This is used to determine boundaries. Typically this is set by the `setBounds` function.

- **hidden**: Boolean
  Flag to determine if an object is hidden or not. This is used by other pieces of the engine to determine if this object should be acted on. For example, a renderer can determine to call `draw` depending on the state of the `hidden` attribute.

- **selected**: Boolean
  Flag to determine if an object has been selected or not. This is useful in interaction with an end-user.

- **container**: `epic.engine.Container`
  This is a handle back to the container for the visualization. This can be used to trigger events that should go to another object or the visualization as a whole.

- **model**: Object
  The underlying data object. This is a consistent place to store data associated with the object.

---

### Methods

- **setPosition(x: Integer, y: Integer, z: Integer): Void**
  Sets the x, y, and z position.

- **setBounds(width: Integer, height: Integer): Void**
  Sets the boundaries of the object. This function assumes a rectangular shape by default. If a more complex shape is needed, a developer must override this function.

- **existsAt(x: Integer, y: Integer)**
  A simple check to determine if an object exists at a given coordinate. This is based on a rectangular shape. If a more complex object is used, this function must be overridden.

- **draw(): Void**
  The main draw function for an object. This is empty by default as the framework does not know how a developer draws to the screen. This must be overridden to provide functionality.
4.3.5.2 User Interface

The epic.engine.Ui objects are used to register user interface elements with various components of the engine. This is a simple object that holds onto the DOM element and a description. This object is intended to be a useful construct in creating complex user interfaces. Developers should subclass this object and extend functionality. The API for the Ui object is as follows:

```
Attributes

element: DOMObject
The underlying DOM object.

description: String
A description of the UI element.
```

4.3.4 Runtime

The epic.engine.Runtime object is one of the two main objects that a developer will be extending. This object, in conjunction with the epic.engine.Renderer object, is how the everything is tied together.

The purpose of this object is to be the heartbeat of a visualization. When the VisualizationManager calls start or stop on a Visualization, the start and stop delegated to the this object. The developer must determine what happens at these two stages in the visualization's life cycle. Typically, a update loop must be turned on or off depending on the state of the visualization. The update loop should be defined within this object.

The update loop should drive the entire application. It should call update on the Renderer and Scheduler, as well as any other necessary objects. Most HTML5 Canvas libraries, such as Processing.js, provide a useful draw function. Developers are encouraged to reuse this functionality.

The API for the epic.engine.Runtime object is as follows:

```
Attributes

scheduler: epic.engine.Scheduler
The scheduler object for the visualization. The Runtime should call tick on this object in its main loop.

renderer: epic.engine.Renderer
The renderer object for the visualization. The Runtime should call update on this object in its main loop.

container: epic.engine.Container
The container object for the visualization. This is useful to have so the runtime can trigger events on the container.

visualization: epic.engine.Visualization
A handle back to the visualization object that contains the Runtime.
```
Methods

init(<init: Dictionary<Object>>): Void
This function is used to initialize the state of the Runtime. This is called by the Visualization it is initialized. This function also registers a resize event with the window that delegates the event to the Container. This function calls setUp.

setUp(): Void
This function is empty by default. A developer should override this function if more initialization is required beyond what is provided by init.

start: Void
This function is empty by default. A developer must override this function to start the Runtime. The Visualization calls this function when it is started. This is left to the developer.

stop: Void
This function is empty by default. A developer must override this function to stop the Runtime. The Visualization object calls this function when it is stopped.

4.3.5 Renderer

In a similar vein as the epic.engine.Runtime object, the functionality of this object is left entirely up to the developer. Since this object is directly related to displaying visual elements, a developer must build this object themselves. The framework is library agnostic and has no knowledge of how a visualization displays elements.

The API provided by the epic.engine.Renderer object is as follows:

Methods

init(): Void
This function is empty by default. The developer should override this function if initialization of the Renderer is necessary. This function is called after the Visualization is initialized.

update: Void
This function is empty by default. The developer should override this function to provide updates to the screen, if necessary. The Runtime should call this every time the main loop executes. This is left to the developer.

4.4 API

The API namespace is an area to keep all code that interacts with third party APIs. This includes APIs that the developer owns and controls to ones that they do not. Typically, these API objects are intended to be used by services. The developer is intended to fill this namespace with useful functionality. An example API is provided and described in the following section.
4.4.1 EPIC API

The EPIC API object is an example API that talks to Project EPIC’s [12] data collection infrastructure. The purpose of this API is to both provide an example and basic functionality for future developers.

This API provides an interface to retrieve a listing of events and a specific Lucene [13] search from the EPIC stack. This can be used to build a visualization using these Lucene searches on specific disaster datasets. The API has the following API:

Attributes

/auth: String
The base 64 encoded authorization string used to talk to Project EPIC’s collection service.

/apiUrl: String
The current URL of where the collection service lives.

Methods

_makeAuth(user: String, password: String): String
Returns the base 64 encoded authorization string.

_setAuth(username: String, password: String): Void
Takes the username and password, encodes them, and stores them for later use. This function must be called prior to getSearch and getEvents.

Call this function to perform a search against the collection service. This function queries EPIC with the provided search string, and upon a successful response, invokes the callback function with the returned data. The params argument contains any http parameters that the code calling wishes to add.

_getEvents(callback: Function): Void
This function queries EPIC’s collection stack for a listing of all known events. Upon a successful response, the function will invoke the callback function with the returned data.

4.5 Utility

The util namespace contains useful functions that do not fit anywhere else. These are functions that are useful to the engine, api, or user specific visualizations. This namespace should be extended by the user. There are a few functions defined, with namespace being the important one, but otherwise contains very little functionality. The intention is to provide a consistent and isolated place to add utility functions.

Users can easily add their own utility functions as they see fit. A user can add their own utility functions in two ways. One way is to open the util.js class and add functions in the same manner as other functions in the file. The added function will be under the util namespace, and can be invoked by calling epic.util.func(), where func is the user-defined function. Users can also add functions in their own file by using the epic.util.namespace function:
Even though this code exists in a file distinct from the framework’s util.js file, the function will fall under the same namespace. This is a useful way to keep all utility functions under the same namespace. If this is not ideal, or the user would like a separation of concerns, they should place their own utility functions under their specific namespace.

### 4.5.1 Namespace

JavaScript lacks language support to cleanly separate your code into different namespaces. Having multiple visualizations living in the same codebase can quickly lead to naming clashes. You can imagine having an object called "node" that is a name for two very different objects, one being an SVG object and another being an eventful canvas object. The way to avoid this is to place things into a namespace.

The namespace library function allows users to namespace their visualization and tools cleanly. This namespace utility is based on similar functionality found in Java. The periods separating names are nested package names. If you have a namespace "EPIC.UTIL.TOOLS" each name, EPIC, UTIL, and TOOLS are distinct namespaces, with TOOLS being a child of UTIL, and UTIL being a child of EPIC. Simply define your methods in this manner:

```javascript
epic.util.namespace("my.utility.namespace", function(namespace) {
    namespace.func = function() {
    }
});

epic.util.namespace("my.utility", function(awesome) {
    awesome.func = function() {
    }
});
```

The namespace function will create any parent namespaces if they do not exist. Note that the function you wish to namespace must be defined on the namespace variable passed into the anonymous function. You now have access to this function anywhere in your code under your defined namespace. For example, to invoke the functions defined in the previous code block:

```javascript
my.utility.namespace.func();
my.utility.func();
```

A namespace can hold functions along side of other namespaces. If a user defines a namespace and then a function with the same name, or visa versa, the behavior is undefined, so careful naming conventions should be exercised. If there is risk of collisions, defining namespaces in all capitals and functions as lowercase is a good solution.
4.5.2 Other Functions

The utility namespace provides a few other useful functions needed by the engine and for Project EPIC specific services. The following functions are provided in the utility namespace:

- **isArray(object)**
  Returns true if an object is of type Array, and false otherwise. This provides a simple way to determine if an object is an array. If the object responds to `toString` with '[object Array]' this function returns true, otherwise it returns false.

- **base64.encode(input)** [16]
  Returns the base64 encoding of the input string.

- **base64.decode(input)** [16]
  Returns the decoded string from a base64 encoded string.

- **dist(u, v)**
  Returns the Euclidean distance between points u and v. These two parameters must have an x and y attributes.

5. Conclusion

This document has outlined a framework that provides the necessary functionality to quickly create useful visualizations and mash-ups. The framework provides several key components that are useful to all types of tools. It also provides constructs to organize and manage complex code structures.

This is particularly useful in the crisis informatics space as the need for tools to aid in disaster situation is becoming urgent. When a disaster strikes, such tools can be of paramount importance to analysis and rescue efforts. Researchers and developers need frameworks to help them build tools quickly and effectively.

Currently, the best platform to create crisis tools is utilizing the browser through JavaScript. The browser provides easy access. A user simply opens a browser, navigates to a page, and benefits from the hard work being done by researchers. With exciting new technologies, such as HTML5, tools and visualizations can be built and accessed with relative ease. The problem with JavaScript is that it is a fairly new ecosystem. There are frameworks starting to emerge, but there is still a fair amount of work to be done. The framework presented in this document attempts to fill some of the gaps observed with these new technologies.

By allowing developers the ability to manage and conceptualize various visualizations within the same framework, while providing useful common functionality, such as event management and task scheduling, I have attempted to ease the pain of adopting JavaScript as a technology for crisis informatics tools. My hope is that future work will be done to formalize and expand this framework to make it even more useful.
Bibliography


Appendix B - Object Oriented JavaScript

The framework was designed using object-oriented principles. This paradigm provides a clean and simple way to use and extend the framework. A very Java-like approach to object creation and inheritance is used. Many visualization engines are implemented using object-oriented principles with a classical inheritance structure. Furthermore, since Processing is a very popular visualization engine, and users want to bring their Java/Processing-based visualizations to the web, they will find Processing.js a natural solution. By building a framework based on the same design principles, I am attempting to make this transition more seamless.

This framework emulates that style as much as possible, with the hopes of reducing the learning curve. JavaScript is a very expressive language that allows developers to pick various paradigms to solve their problems. However, when building a visualization, most researchers care about the end product, not the style they used to get there. By enforcing a strict object-oriented approach to visualization design, the framework encourages good practices and style, so a researcher can focus on their research, not the language. By implementing an object-oriented paradigm in the design of the framework, there is a clean separation of concerns among the components.

For the sake of a user implementing an idea using this framework, I will present examples that use real objects found within the framework. The specifics of each object will be explained in Section 4, where each component of the framework is described in detail.

I chose to avoid the new keyword, as it is difficult and prone to mistakes with developers new to the language. Instead, to create an instance of an object within the framework, developers will simply call a constructor function. The following is an example of creating an eventful object:

```javascript
var eventful = epic.engine.Eventful();
```

This is all the code required to create new object. If the constructor takes an initialization object or parameters, the code looks very similar. Here is an example creating a `Container` object that takes a `VisualizationManager` as a constructor argument:

```javascript
var manager = epic.engine.VisualizationManager();
var container = epic.engine.Container(manager);
```

Object inheritance is a bit more complicated than simply calling a function. It works differently than Java flavored languages. In order to create an object, which inherits from another, the prototype chain must be set properly. The key component to this style was using the relatively new object creation method defined in ECMAScript 5, `Object.create`.

Object.create

Unfortunately, there are many ways to define objects in JavaScript. Since JavaScript is a prototype-based language, inheritance works by pointing an object’s prototype to the parent object’s prototype chain. In order to simulate traditional inheritance, a user of the framework should inherit a parent object’s prototype chain using `Object.create`. It is important to remember that there is no concept of a class in JavaScript. There are only objects, and objects that inherit from other objects, meaning their prototype chain points to their parent’s prototype chain.
This pattern employs what’s known as differential inheritance. The idea behind this type of inheritance is that most objects are the same. When you inherit from another object, you change a few properties, or extend the previous object, but for the most part are the same object. Differential inheritance allows each object to reference the same prototype chain, but contain their own distinct table of properties [17]. Often, you only need to initialize a property differently to distinguish an object from its parent. This inheritance model makes this very straightforward. For the problem this framework is tackling, differential inheritance is the best solution.

If a user needs to create an object with no parent object, the function call to `Object.create` is straightforward:

```javascript
var object = Object.create({});
```

If a user wants to create a simple object that has a method, simply passing an object with functions defined will do the job:

```javascript
var myObject = Object.create({
    memberFunction: function() { }
});
```

One may ask why go to the extra work to call `Object.create` when the following code snippet is functionally equivalent:

```javascript
var myObject = {
    memberFunction: function() { }
};
```

The reason for defining objects using the `Object.create` is it provides a useful and consistent way to inherit from other objects. For example, if you wanted an eventful object with added functionality, the following achieves this:

```javascript
var myObject = Object.create(epic.engine.Eventful());
myObject.memberFunction = function() { };
```

This code creates a new `epic.engine.Eventful` object, creates an object with its prototype chain pointing to the eventful object, and assigns `myObject` to point to the new object. If you needed to inherit from an object that had some initialization parameters, this works as expected:

```javascript
var manager = epic.engine.VisualizationManager();
var myObject = Object.create(epic.engine.Container(manager));
myObject.memberFunction = function() { };
```

Notice the `manager` object being passed into the constructor function of the `epic.engine.Container`. To add member variables to an object, you can pass a second (optional) parameter, called the `propertiesObject`, to `Object.create`:
In this example, you can access the member variable `my` calling `myObject.memberVariable`. A member function can access the same variable by qualifying the variable with the `this` keyword.

One important aspect to defining member variables is how you define the `propertiesObject`. Notice that you don't assign the value of `memberVariable` directly, but pass it a hash containing the `key` `value`. When the JavaScript engine creates the object and reads the `propertiesObject`, the initial value is defined to be the object found under the `value` key.

To convention for defining a private member function or variable is to prefix it with a “_” character. This does not prevent access, however. If a user wants to be explicit, and truly prevent access they can define private member functions or variables in the following manner:

```javascript
var _privateMemberVariable = 0;
var _privateMemberFunction = function () {
    return _privateMemberVariable;
};

var myObject = Object.create({
    _privateMemberFunction: function () {
        return this._privateMemberVariable;
    },
    memberFunction: function() {
        return this.memberVariable;
    }
}, {
    _privateMemberVariable: { value: 0 },
    memberVariable: { value: 0 }
});
```

One important difference in building private variables and functions in this manner is the use of the `this` keyword. If the declarations exist outside of `Object.create` then calling either the function or variable should **not** be prefixed with the this keyword. While a user is free to create objects how the wish, the suggested pattern is to define private variables and functions inline with public parts. Doing so keeps member access consistent across public and private attributes.

The reason for this syntax is the member variable has other properties associated with it. Omitting
these properties causes the member variable to act different than a traditional JavaScript developer is used to. Using the previous example, if a user wishes to have `memberVariable` that acts like a traditional JavaScript attribute, defined in ECMAScript 3, create the `propertiesObject` as such:

```javascript
var myObject = Object.create({
    memberFunction: function() {
        return this.memberVariable
    },
}, {
    memberVariable: {
        value: 0,
        writable: true,
        enumerable: true,
        configurable: true
    }
});
```

Note the three extra properties passed in; `writable`, `enumerable`, and `configurable`. Also, this can be used to initialize or overwrite inherited member variables. To find out more regarding the use of the `propertiesObject`, refer to the Mozilla JavaScript Reference [17].

The pattern for creating a constructor function that returns a new object is simply a function, which returns an object, built with `Object.create`. Constructor functions should be capitalized to distinguish them from normal functions. The following code snippet illustrates this:

```javascript
var Constructor = function() {
    return Object.create({
};
}
```

Now you can use this constructor like so:

```javascript
var myObject = Constructor();
```

All objects within the framework are created using this pattern. Users of the framework are strongly encouraged to be consistent and follow this pattern. Putting everything together, here is a full working example that shows a constructor that takes an initialization hash, builds an object that inherits from another object, and defines a member function and member variable:
This syntax may appear confusing at first, but once a user gets used to it, it provides a clean and straightforward way to do inheritance in JavaScript, while avoiding the problems with other patterns. The pattern described above is how all objects are defined within the framework.