User Web-Browser Event Visualizer

A Major Qualifying Project

Submitted to the Faculty of Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree in Bachelor of Science

in

Computer Science

By

Luke Ludington

Ruyue Wang

Da Xu

Date: March 28, 2020

Sponsoring Organization: Shape Security

Project Advisor: Mark Claypool

Shape Security Advisors: Michael Ficarra, Ying Chau Mak

Abstract

Shape Security is a cybersecurity company that provides solutions that reduce bots, fraud, and unwanted automation. To assist its data scientists to better understand bots and fraud, Shape has developed multiple data visualization tools such as VizMonkey. VizMonkey transforms collected user input signal data in a transaction into a visualization; this enables its users to provide quick manual analysis of a specific transaction. However, Vizmonkey lacked the ability to visualize much of the data Shape has and lacks features that would be helpful for the expanded scope of its use. Thus, our goal was to develop a new version of Vizmonkey which was more user friendly, had better representations of the data for non-experts, and was able to display all the data Shape collects about user events. After two months of design, development, and testing, we showed the new tool to the team that will directly use it at Shape and iterated over received feedback. The new Vizmonkey should help Shape perform manual analysis more quickly and easily and produce better protection for customers' websites.

Acknowledgments

The success of our Major Qualifying project (MQP) would not have been possible without the help of many individuals. We would like to start off by thanking our sponsor, Shape Security, for the chance to contribute to the internal tool and the resources required. Additionally, we would like to thank our mentor, Michael Ficarra for constantly providing guidance throughout the development process and for defining our project parameters before we started. Furthermore, we would like to thank our MQP advisor Mark Claypool for his support during the project and guidance in writing this paper. Lastly, we would like to express our gratitude towards our school, Worcester Polytechnic Institute (WPI) for giving us this amazing opportunity.

Table of Contents

Table of Figures

1 Introduction

Shape Security, a cyber security company in Santa Clara, California, works with many different data sets. It analyzes these data to help companies detect and extinguish interactions with automated intruders (bots); these intruders use up valuable server resources for clients and can lead to data breaches and account theft. Bots have become more and more sophisticated in their attempts to mimic genuine human interaction. In order to explain the inner machinations of Shape's protection solutions to its clients, as well as to allow the engineers to see the data and develop better algorithms around it, Shape has made many visualizers to help interpret its data.

Visualizers are a common and effective tool for data analysis. Different kinds of data are collected within all industries collected, various data visualizers start to be designed. Those visualizers aim to assist people to get a better understanding of complicated relations within data. For example, a visualizer could be created to help scientists to understand patterns from scientific data which are not visible in the real world like a gravitational field. Another visualizer could be one which shows multiple video streams of the original data on top of each other to represent that data as closely as possible to how it was initially captured. Visualizers can aim to interpret abstract data or show as closely to its source as possible.

Given that visualizers can help discover trends and patterns for people, Shape Security built a user event visualizer called VizMonkey 1. This visualizer transforms the collected user input signal data to visualizations and enables Shape to perform manual analysis of user interaction data more quickly than by looking at the raw data.

VizMonkey 1 renders a visualization of a user's interactions with a given website by reading selected user input signal data. Signal data is stored in a JSON file that contains various types of user interaction data such as keyboard mouse events. Keyboard events represent a user's

interaction with the keyboard such as pressing or releasing a key. Mouse events record a mouse movement path of the user's input as well as any mouse clicks the user performs.

VizMonkey 1 lacked usability and feature issues with the current visualizer. Firstly, it was very limited and required instruction to interpret its output. New hires and clients had to spend more time getting used to this tool, resulting in inefficient onboarding process and ineffectual use of the tool. Second, it did not have the ability to visualize much of the data the company collects and lacked features that would be helpful for expanding scope of its use from a purely internal tool to one that could be used by clients directly.

The goal of our project was to redesign VizMonkey 1. With the new tool built and deployed, Shape Intelligence Center (SIC) team members will spend less time on understanding user behaviors and will be faster in creating accurate determinations. Also, the new renderings will be more professional looking and can be more easily shared with current and prospective customers. The end result of the project was to provide a quick and easy manual analysis tool for the employees in Shape Security and make sure that their customers can get much more professional analysis reports, which was deployed and is currently in use.

2 Background

SIC's main function is to show Shape's current and prospective customers the value that Shape's provide. SIC does this demonstrating the automated traffic that Shape has blocked from using their services and the way Shape was able to identify and block that traffic. SIC uses data visualization tools to provide this service. I

2.1 VizMonkey 1

VizMonkey 1 was designed to view data collected by Shape Client. Shape Client collects data about user interactions with websites that Shape protects. Among the data it collects are those representing a user's mouse and keyboard inputs; screen, window, and viewport dimensions; and system configurations including operating system and browser. These data are used by VizMonkey to approximate what a user's interaction with a website may have looked like.

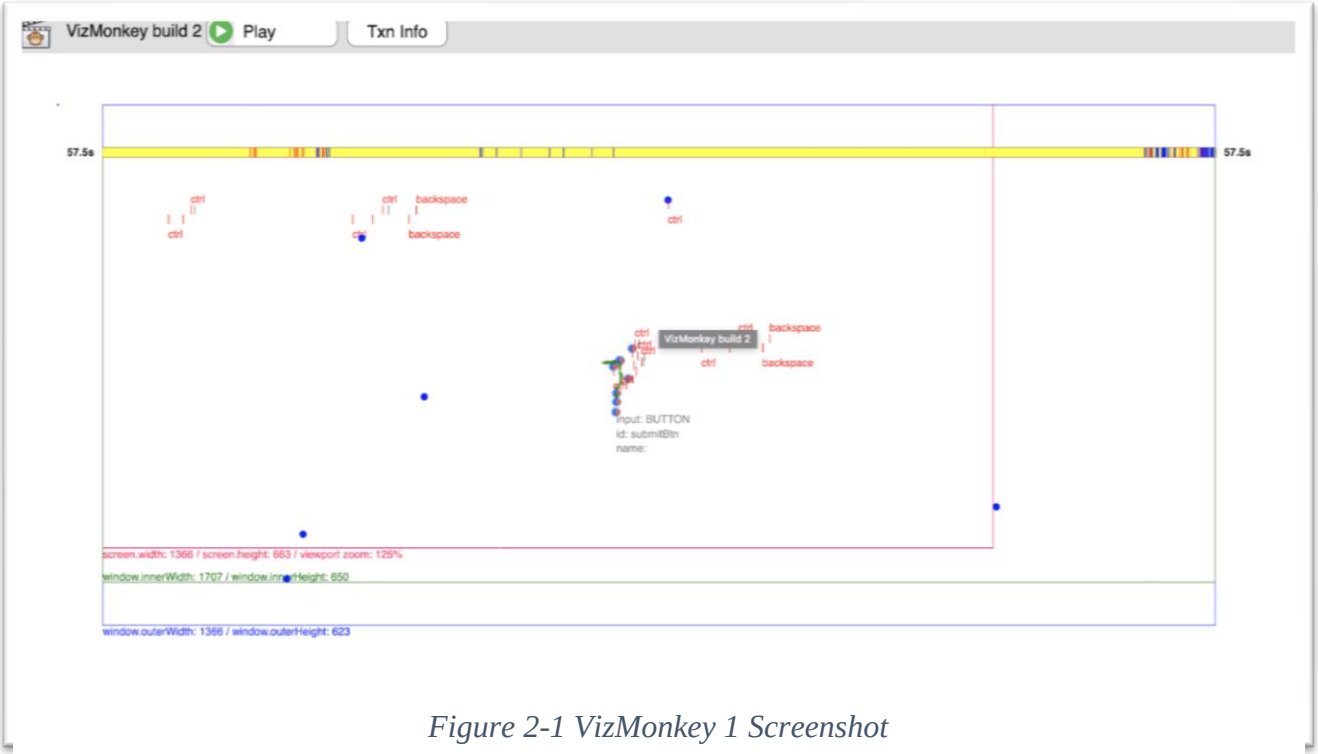


Figure 2-1 VizMonkey 1 Screenshot

2.1.1 Capabilities

VizMonkey 1 (depicted in Figure 2-1) shows mouse data and keyboard data as events on the screen. The mouse data are shown by the red, blue, and green dots on the screen. The blue dots are mouse click events with the blue circles around them representing mouse down. The green dots are mouse move events. Lastly, the red dots are mouse up events.

The screen, window and viewport dimensions are depicted by displaying 3 differently colored rectangles at their respective resolutions all anchored in the upper right-hand corner of the playback area. The red rectangle represents the screen's dimensions, the green rectangle represents the viewport's dimensions, and the blue rectangle represents the window's dimensions.

A keyboard event is any key being pressed. These events are represented in 3 parts; each part is represented with a red line and the key being pressed (this is not always available as printable keys are not recorded for privacy reasons). The first part of a keyboard evert is the key-down event, followed by the key-press and key-up events. The location of the keystrokes on the screen is loosely related to where the mouse cursor was at the time of input.

All of these data are replayed together to show an animated view of what the interaction looked like. There is a timeline at the top of the playback which shows vertical lines for each event. These lines are also colored, red for keyboard events and blue for mouse events.

2.1.2 Problems

VizMonkey 1 represents most of the data, but in a way that is largely unintuitive and difficult to use. First, although the VizMonkey1 can play a user transaction, it lacks the ability to scrub through that transaction after playback has started. This makes looking at longer transactions laborious as a playback must get to the time of an event on its own before the event

can be viewed. It also lacks the ability to pause once that time has been reached, so a transaction cannot be viewed a specific point in time.

Second, VizMonkey1 requires detailed instructions to interpret its output; for instance, it reproduces the dimensions of the screen, window and viewport in the same place with colored rectangles showing them at their reported resolution. Showing them in this way means that is hard to notice when the viewport is larger than the screen for inexperienced users. It can be easy to not notice that one colored square is larger than its "bounding" square and get them confused for each other. This method of screen representation also does not show the location of the browser on the screen.

Thirdly. the keyboard data are represented in their raw form which requires the user to try and interpret which key ups, downs and presses go together. This also does not convey information about how long a key was pressed for. The current data representation is not only hard for non-experts, such as shape current and potential customers to understand but also difficult for internal employees to analyze. When talking to a SIC employee about a specific transaction it took us several minutes to realize we had been misinterpreting which resolution was which despite the fact that the employee works with VizMonkey every day.

Lastly, VizMonkey 1 showed older events on the left and newer events on the right, with the Y-axis having no meaning (there were height changes, but they didn't represent anything) the distance between events also had no meaning. This required people to make the connection between the timeline and individual events which could only really happen during playback, meaning that no information was displayed statically. It was also an underutilization of screen real-estate.

Some of the data representations are inaccurate in VizMonkey 1. For example, the current Shape Client data does not record the positions of keyboard events but VizMonkey1 attempts to put the keyboard events within the screen area, where it thinks the input was. This lead to nearly everyone on the SIC thinking that the Shape Client could record this information and requesting that we include it in VizMonkey2.

2.2 Requirements

Shape, upon assessment and use of VizMonkey 1, created a set of features which VizMonkey 1 already had and features they wanted to add.

Main goals:

- Display the dimensions and position of screen, window, and viewport
- The ability to switch between physical size and scaled-to-fit
- Speed-control and a scrubber with the end time of the event and indications of events
- Attachment of OS-appropriate mouse cursor on the mouse movement path
- Visualization of mouse clicks and touch events
- Visualization of record of keyboard activity (which keys were pressed)
- Visualization of mouse wheel events
- Visualization of visibility of the window
- Google Cloud deployment visible to Shape Security employees
- Documentation of development and release process, with a clear user manual

Stretch goals:

- A log-style chronological textual rendering of the events in one big list or table statistics for key-down intervals, key down/up timing, etc.
- The ability to export the renderings in video format

- The ability to play multiple transactions at once
- Heat map representation of positional activity in multiple transactions

We were also given a mockup of VizMonkey 2 might look like by Shape:

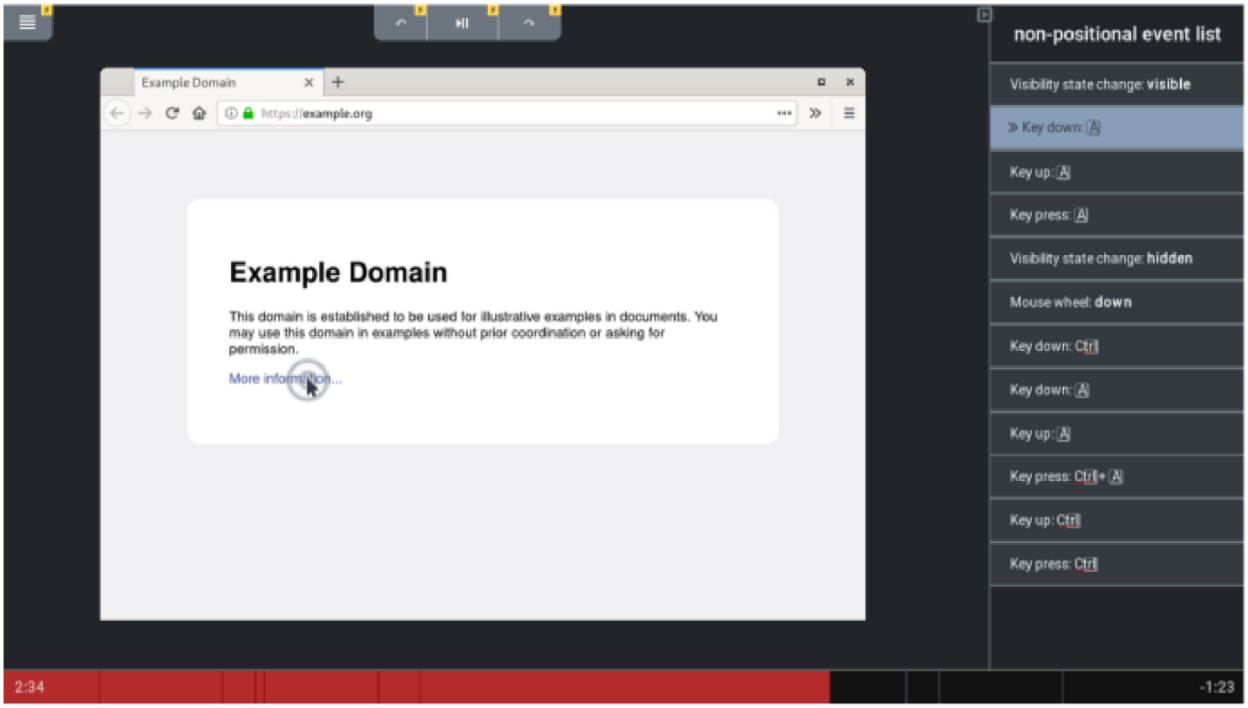


Figure 2-2 Shape VizMonkey 2 Mockup

This mockup shows a more realistic browser window and mouse interaction with a section dedicated to non-positional, events like keyboard events.

2.3 Technology Stack

To create a web application to replace VizMonkey 1, the three main technologies we used were React for the general architecture of the application and the user interface components; Redux to allow for the components of the application to talk to each other and allow for the application to save its data locally; and AnimeJS to allow for the creation of animations and to link those animations together on a unified timeline.

2.3.1 React

React is an open-source JavaScript library that is used for building user interfaces specifically for single-page applications [1]. It's used for handling the front end of web and mobile apps. There are multiple advantages to using the React framework instead of other modern front-end frameworks. React is easier to use than AngularJS and ViewJS because of its component-based development approach and well-defined lifecycle, and therefore can be used to build a professional web application within a relatively short time. React uses a one-way data-flow which makes maintenance much easier for future developers as it allows developers to see why a specific component is in a certain state by looking only at its parents not it siblings.

VizMonkey 2 is integrated into a larger internal application with SIC called SIC tools. SIC tools was made in React—this was another motivation behind using React as this made the integration process with SIC tools much easier. Due to the hierarchical nature of React VizMonkey can be encapsulated into a single component that can then be loaded within the SIC tools application.

2.3.2 Redux

Redux is a "predictable state container for JavaScript apps" [2]. In other words, Redux makes it easier to manage the state of the web application. Due to the complexity of VizMonkey 2, multiple components need to interact with one another to fulfill one functionality. Also given the fact that React only supports one-way data flow, the optimal solution is to use a state management tool that stores all the states in the program, with easy accessibility through all the components in the application. Redux serves this purpose and thus we choose it to manage the states in our application.

Redux has individual components dispatch events to the store. When the store is updated the components which are based on those data are updated. After a component dispatches an event it goes through the appropriate reducers and then changes the state of the store. When the store's state is changed any component listening to that data updates.

2.3.3 AnimeJS

Anime.js is described as a "lightweight JavaScript animation library with a simple, yet powerful API". AnimeJS works with CSS properties, SVG, DOM attributes and JavaScript Objects [3]. There are lots of features packed into Anime.js such as keyframes, timelines, and SVG animation.

Anime.js has all of the features necessary for basic keyframe animation in Javascript. First, Anime.js provides the timeline. It is easy to maintain relationships between different animations by using the timeline. To use the timeline and event is create, such as moving a DOM element or drawing an SVG line and that event is given a start time and a duration then place on the timeline. Second, Anime.js provides built-in callback and control functions; there are play, pause, control, reverse, seek and trigger events, which can be used to easily manage and control the state of animations [3]. Moreover, Anime.js supports the majority of modern browsers including Chrome, IE/Edge, Safari, and Firefox. In addition, Anime.js has well-organized documentation that demonstrates HTML, JS codes and great examples, which help speed up the progress of learning Anime.js. In short, Anime.js is versatile, powerful, and relatively easy to learn and use.

2.3.4 Scalable Vector Graphics (SVG)

SVG is a markup language for describing two-dimensional graphics applications and images [4]. The current prototype uses SVG to help draw the mouse movement path.

There are several advantages for using SVG. First, SVG has good scalability, therefore it can be printed with high quality at any resolution. Second, because SVGs are rendered in XML (Extensible Markup Language), the file size is relatively small, which provides a boost in load time and performance. Moreover, SVG is supported by most modern browsers [4].

However, there are several disadvantages of using SVG. The major disadvantage is the learning curve of SVG is a little bit steep. Because SVG is structured XML, it is not easy to understand the code behind it. Furthermore, the codes of SVG can be extremely complex and lengthy, which is difficult to debug.

3 Design Process

When analyzing VizMonkey 1 we noticed that the principal problems with it were all due to its user experience, this is why we started designing VizMonkey 2 from a user experience perspective. Instead of looking at the data and defining a representation that was most closely related to the it, we thought about what the data meant and designed a representation that was most closely related to the way people understand those things.

3.1 Design of Keyboard Representation

When using VizMonkey 1, the most confusing data representation was the representation for keyboard input so that is the first place we decided to improve. We thought that most people would think of keyboard event as a single key going down then that same key going back; this combination of events is called a keystroke. This is different from how Shape Client records keyboard data. Shape Client records as a series of instantaneous events representing key ups, press and downs—this is an artifact of the way JavaScript records key events. These events are then stored as an objects which contain the time of the event relative to the start of the recording, the input area of the event, and the key that generated the event except for printable keys which are masked to protect privacy. VizMonkey 1 represented keyboard data as the raw key ups, presses, and downs for VizMonkey 2 we represented them as keystrokes with their width representing the duration of the event.

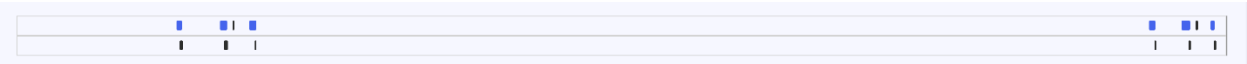


Figure 3-1 Key Stroke Panel

Once a keystroke is defined it still must be shown on the screen. In order to encode meaning in the placement of the keystrokes on the screen keystrokes are organized temporally

with their location representing the time they occurred; this was a natural extension of the keystroke's width representing the duration of a keystroke. These keystrokes were put together in an area called the keystroke panel.

Keystrokes have a duration therefore it is possible that multiple keystrokes will occur at the same time. In order to represent multiple keystrokes occurring in overlapping times a keystroke which starts during another will be place lower than it and the keystroke panel will be as tall as the highest number of simultaneous keystrokes.

The type of key each keystroke represented was the most important feature of the keystroke and in order to display this information the color of each key was changed in to be one of several representing shift, tab, ctrl, and alt.

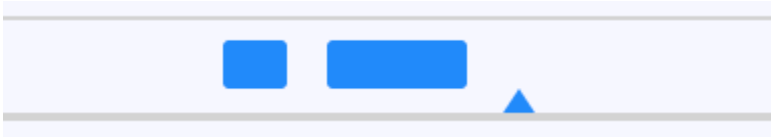


Figure 3-2 Key up

There were some key events that could not be paired with others, this was due to masking of data as well as some events simply lacking a pair in the data. When a key event could not be paired it was represented as a simultaneous event as a up arrow for up, a down arrow for down or a circle for press.

3.2 Design of Mouse Component

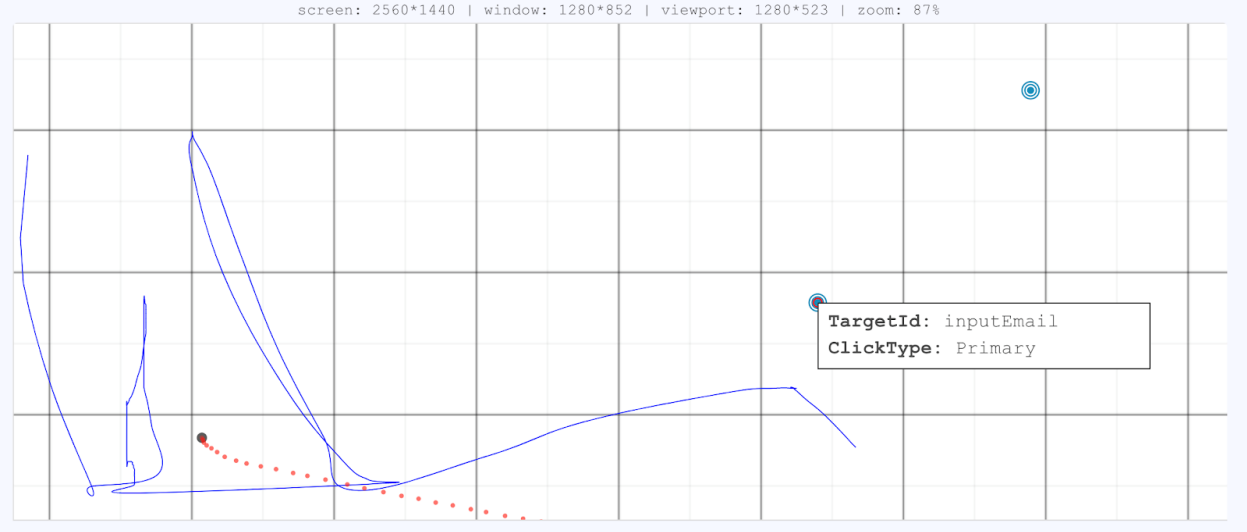


Figure 3-3 Mouse Representation

The Mouse component was represented as a combination of trails and lines. The lines were between the reported locations of the mouse and the cursor followed the lines. Click events were represented as circles, with the largest being down, then up and the center dot being click, these were separated into separate representations because they could occur at separate locations, they most commonly appear together so they needed to be able to overlap in location while still being clearly visible and distinct.

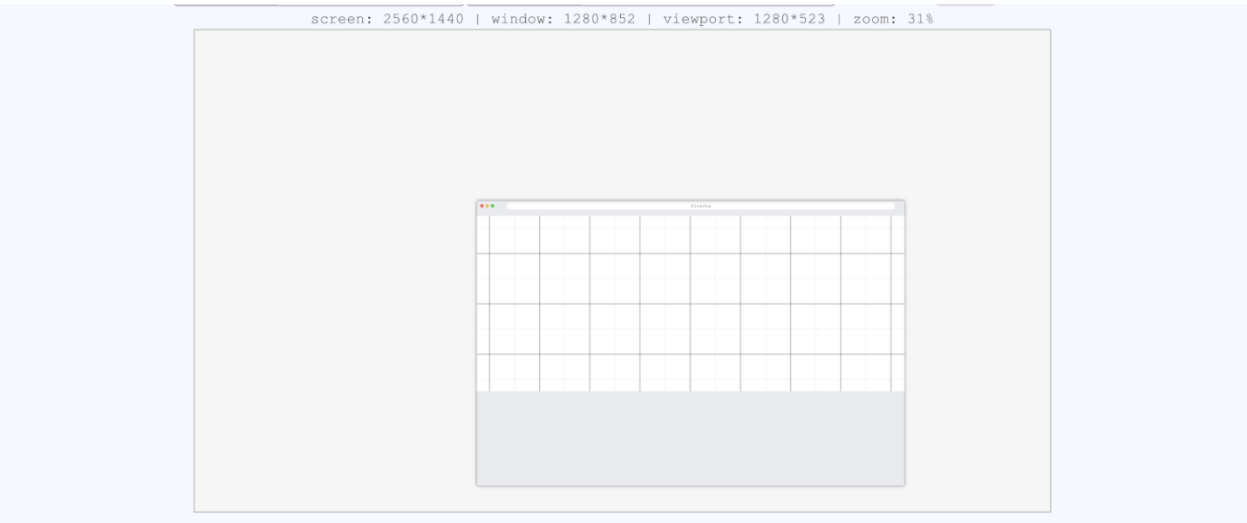


Figure 3-4 Screen Representation

The mouse input was collected with respect to the viewport and only collected within the viewport. The position and size of the window and screen were also recorded, this was enough information to recreate the users screen with some degree of accuracy, so rather than reinterpreting those data we elected to show them as closely to the user's screen as possible with a generic browser representation and a viewport within it. The viewport would be placed at the top of the screen with excessive padding placed at the bottom because normally padding that large would be indicative of a console being open so that was the interpretation we used. The use of a screen like representation made it much easier to understand as it was exactly what the user would be used to seeing.

3.3 Design of Side Panel

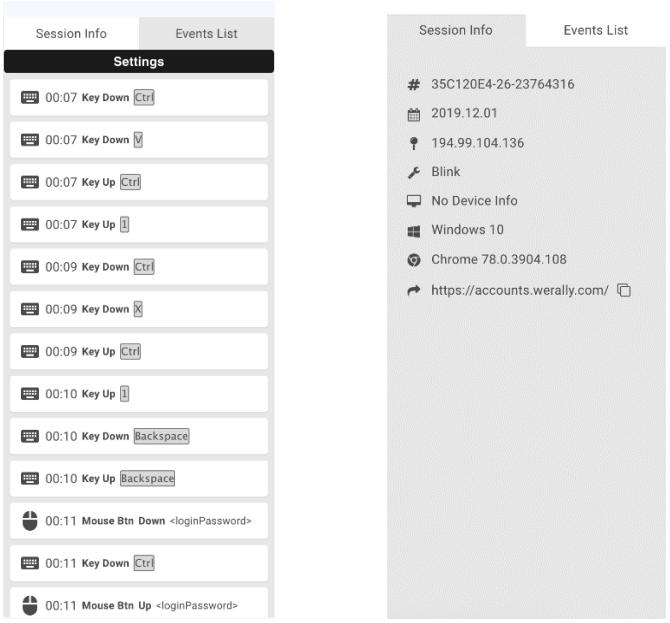


Figure 3-5 Side Panel with Event List (left) and Session Info (Right)

The Side Panel is comprised of the event list and Session Info these were both accessible as tabs within the Side Panel. The Event List was conceived to be able to show more information about a specific event, such as the input are of a click or keystroke, without having to hover over the event in its representation area. This needed to be displayed in a different area from the keystroke and screen areas because the dimensions of those areas and the representations within them were derived from the data and not guaranteed to be large enough to comfortably display long strings of text. In order to create an easy visual link between the event list and the information and the rest of the playback, every event becomes highlighted as it occurs, this also allows for a user to pause and not have to search for the extended information about the event they just saw. The Session Info section of the Side Panel shows the static information about an event such as the browser. This information needed to be accessible to the user, but it is not necessary to have it always visible,

3.4 Design of Menu Component

The menu was designed to be as similar to common office applications as the drop-down paradigm is familiar to almost anyone who has used a computer and was effective at organizing our application's features. Most menus also have quick access areas for commonly used features. our application also has a number of features we expected users to interact with more often than others. These features were specifically about interactions with the screen such as changing the focus of the screen area between the screen, window, and viewport or hiding and showing the side panel. The menu was is mostly static only changing state to highlight the currently active preferences of the application.

3.5 Design of Timeline Component

Figure 3-6 Timeline Component

In order to show any section of a playback we wanted to be able to have a way to interacted with a playback and to allow for jumping to a specific point in time. A common application for all of these features is in video playback, so to create out timeline we looked at common video players like YouTube and looked at all of the features they have and how they represented them. We noted that playback speed, play/pause, skip forward, scrubbing, and restart were among the features they shared, as well our users requested a feature to jump to the next event.

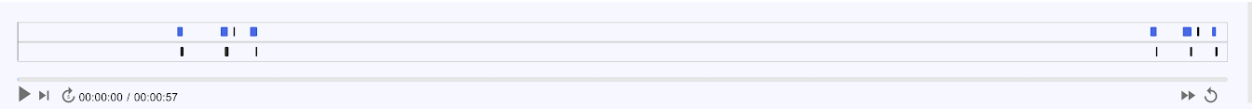


Figure 3-7 Timeline component next to Keystroke Panel

When making the Timeline we placed the scrubber the controls themselves as it was linked to a cursor in the keystroke panel above it. In order to make that link visually obvious those elements were placed next to each other. Beneath the scrubber the timelines controls were placed in two groups. The group on the left, containing the play/pause button, the jump to next event button, the skip forward 5 seconds button, and a display to show the current time of the playback next to the total time of the playback. The group on the right contained the speed selection and the restart button.

4 Development and Implementation

This section dives into the implementation of the parsers and components, which are the two main parts of the application. The parsers and components are written in JavaScript using the React framework.

4.1 Data Parser Implementation

The data source of our application is provided by Shape Client. Shape Client is an application embedded into Shape's customers' webpages that collects signal data to better protect Shape's customers. There are five types of signal data that are used within VizMonkey 2 provided by Shape Client: mouse event data, keyboard event data, screen data, touch event data, and visibility event data. The following sections describe the data parsing process for the signal data.

4.1.1 Keyboard Event Data Parser

Keyboard event data parser is the most important part in keyboard visualization since the way Vizmonkey 2 parses the data determines the way data is displayed. The keyboard parser is mainly contained by three parts: normalization, keystroke formation and keystroke grouping. The following sections explain these three parts one by one.

4.1.1.1 Keyboard Event Normalization

Shape Client collects 3 types of keyboard events. As shown in the *Figure 4-1*, every type of keyboard data is slightly different from one another.

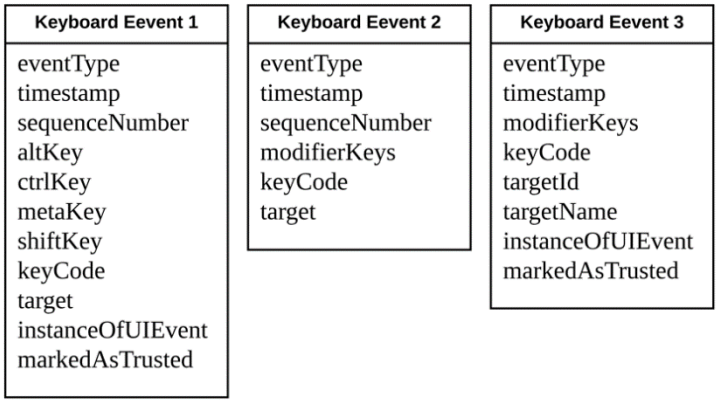


Figure 4-1 Three Types of Keyboard Events

In order to make sure the data input into the keyboard parser is consistent, a normalization function was made to unify these incongruent data into one shape. *Figure 4-2* shows the fields contained in the normalized keyboard event.

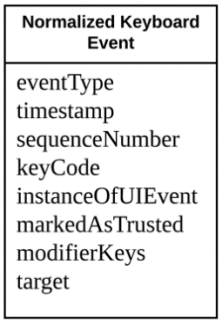


Figure 4-2 Normalized Keyboard Event

The `eventType` is an integer indicating the keyboard event: 1 indicates key down, 2 indicates key up and 3 indicates key press. `timestamp` is an integer (in milliseconds) indicating the start time of the event. `sequenceNumber` is an integer used to associate the key down, key up and key press events. `modifierKeys` represents whether the "ctrl", "meta", "alt", or "shift" modifier key (respectively) was active. t`arget` is the name of the HTML tag where keyboard events took place. For instance, target can be `loginEmail` or `password`.

4.1.1.2 Keystroke Formation

A map is created to group the events according to the sequence number. The key of the map is sequence number and the value is the list of events that have the same sequence number. As shown in *Figure 4-3*, after the mapping, the key down event and the key up event for a Ctrl key event are grouped together, and the key down event and the key up event for a V key event are grouped together.

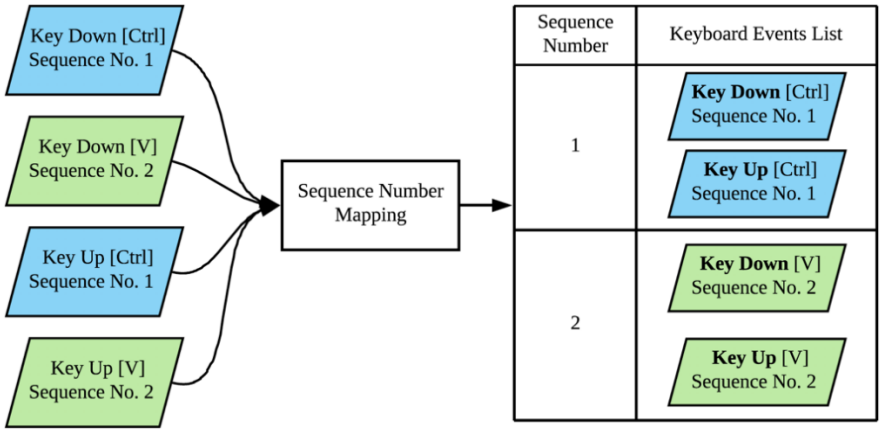


Figure 4-3 Normalized Keyboard Events Mapping

The next step is squashing the keyboard events with the same sequence number into a "keystroke". *Figure 4-4* shows the simplified procedure of keystroke formation.

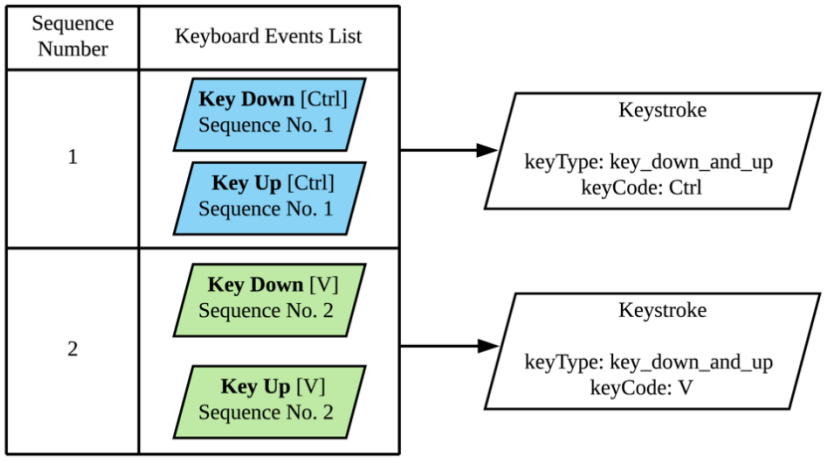


Figure 4-4 Simplified Keystroke Formation

In a normal case, events within a specific sequence number have key down at the beginning and key up at the end, therefore the keystroke is marked as “key_down_and_up”. However, sometimes there are cases where this pairing cannot be made. When this happens, the keystroke is marked as “key_down” or “key_up” depending on which type of event is left unpaired. *Figure 4-5* shows the keystroke formation for a single key down or key up event.

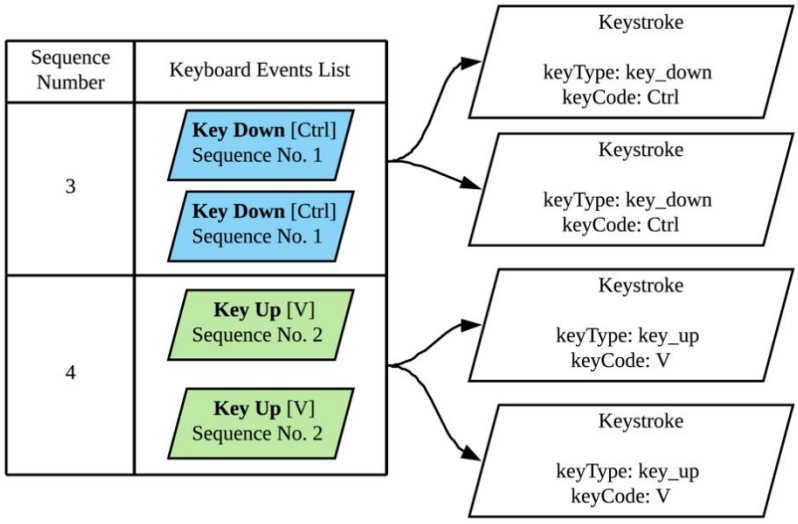


Figure 4-5 Keystroke Formation for Key Downs and Key Ups

Besides the key type and key code, the keystroke object also has some other fields. As shown in *Figure 4-6*, keystroke has start timestamp, end timestamp and target. The start timestamp represents the start time of the keystroke. End timestamp represents the end time of the keystroke. The keystroke has a key down event at the beginning and a key up event at the end, the start timestamp is the timestamp of the key down event and the end timestamp is the timestamp of the final key up event. For keystrokes which only have a key down or key up, the start time and end time are the same, which is the timestamp of the key down event or key up event. The target of the keystroke is the same as the target of the key down event and/or key up event which formed it.

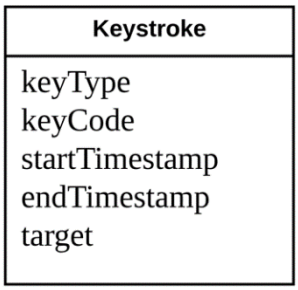


Figure 4-6 Keystroke Object

4.1.1.3 Keystroke Grouping

The final step is grouping the keystrokes according to the start timestamp and end timestamp. It is desired to show all the keystrokes on the screen without having them overlap with one another. In other words, if the start timestamp of a keystroke is larger than the end timestamp of another keystroke, then the two keystrokes should be on the same layer. But if the start timestamp of a keystroke is larger than the start timestamp of another keystroke but smaller than the end timestamp of that same keystroke, the two keystrokes should be put on different layers. *Figure 4-7* and *Figure 4-8* illustrate the two situations. As shown in *Figure 4-7*, the start timestamp of keystroke 2 is larger than keystroke 1 and thus they are aligned on the same layer. In *Figure 4-8* the start timestamp of keystroke 2 locates within the time range of keystroke 1 and therefore keystroke 2 is put on a second layer.

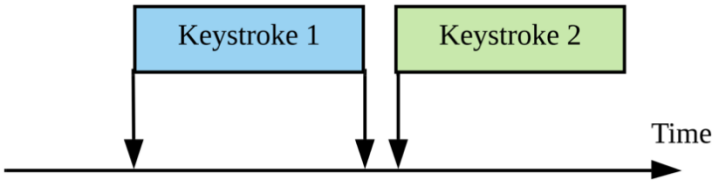


Figure 4-7 Keystrokes on the Same Layer

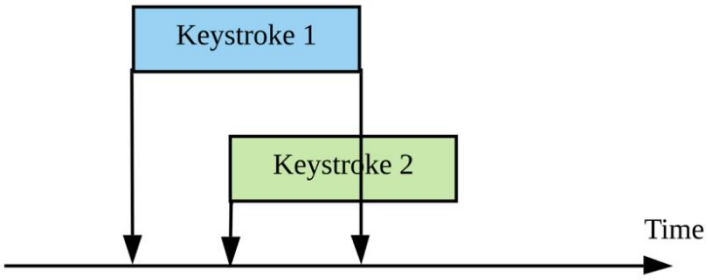


Figure 4-8 Keystrokes on Different Layers

After all the keystrokes are processed by the algorithm above, a list of keystroke groups is generated. An example of this procedure is shown in *Figure 4-9*. Assuming that all of the keystrokes are sorted by the start timestamp before running the grouping algorithm, and assuming that the start time of keystroke 2 lays between the start time and end time of keystroke 1, the start time of keystroke 4 lays between the start time and end time of keystroke 3, the start time of keystroke 5 lays between the start time and end time of keystroke 4, the final parsed keystroke data is formed as the keystroke group list shown in *Figure 4-9*.

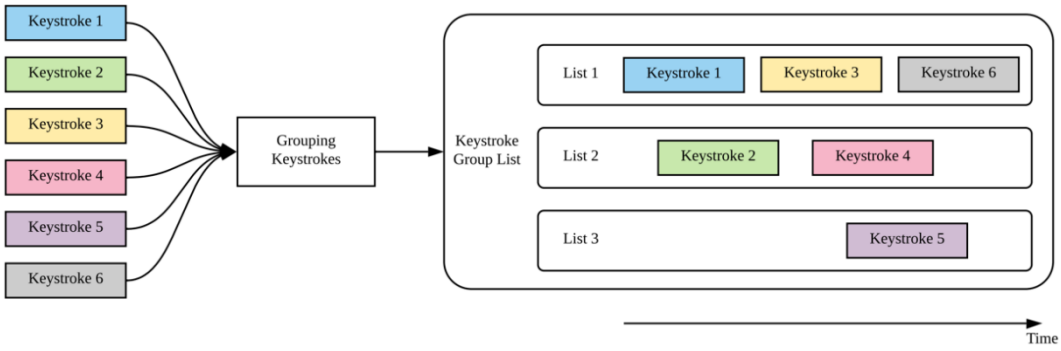


Figure 4-9 Example Keystroke Group Formation

4.1.2 Mouse Event Data Parser

Mouse event includes the mouse button event and the mouse move event. Mouse event data parser contains two parsers, one for mouse button and one for mouse move. The following section discusses types of mouse event and how the mouse data is normalized and parsed.

4.1.2.1 Types of Mouse Event

Similar to the keyboard events, Shape Client also collects three types of mouse events: `mouseEvents`, `mouseEvent2` and `mousEvent3`. Each type of event is slightly different. Specifically, `mouseEvents` contains four parts: `mouseDown`, `mouseUp`, `mouseClick` and `mouseMove`. `mouseDown`, `mouseUp` and `mouseClick` are events related to mouse button and `mouseMove` is related to mouse movement. `mouseEvents2` and `mouseEvent3` are similar, but having three parts for mouse button events, they have a single array called `mouseButtonEvents` and use `eventType` to distinguish mouse down, up, and click. Beside `mouseButtonEvents`, they also have `mouseMoveEvents` that contains all the mouse movement data. *Figure 4-10* shows the different types of mouse events.

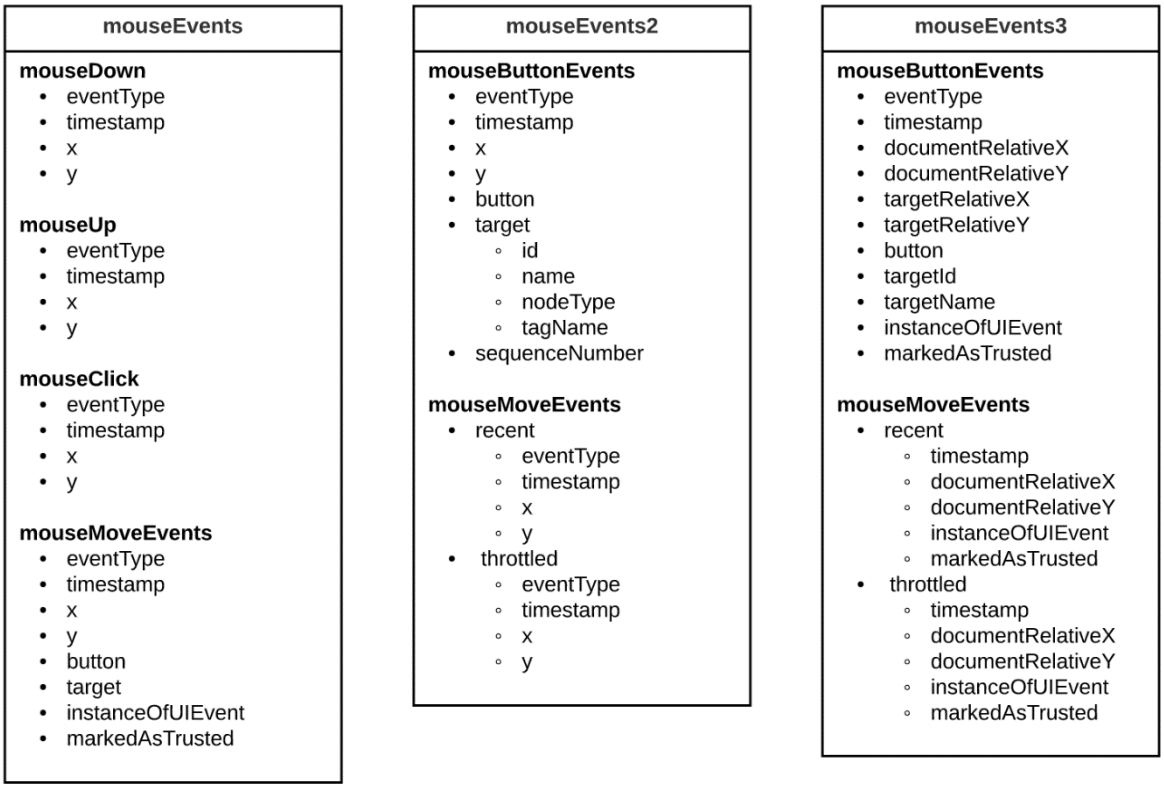


Figure 4-10 Three Types of Mouse Event

4.1.2.2 Mouse Event Normalization

Normalization functions for mouse movement and mouse buttons are created separately. *Figure 4-11* shows the normalized mouse move and button events. Normalized Mouse move and button events have some common fields. For example, `eventType` is an integer indicating the type of event: 1 is mouse down, 2 is mouse up, 3 is mouse click and 4 is mouse move. `timestamp` is an integer (in milliseconds) indicating the start time of the event.

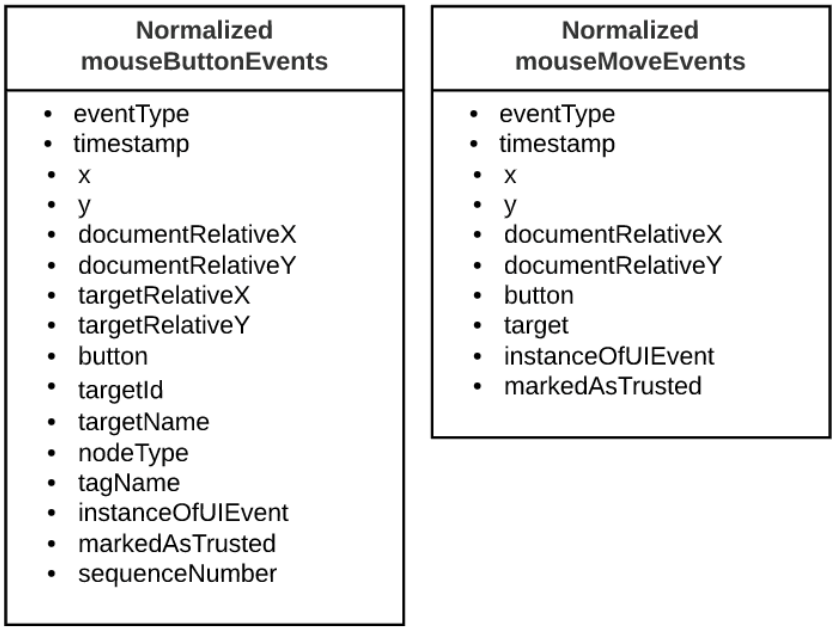


Figure 4-11 Normalized Mouse Button and Move Events

Once the normalized events are generated, a parser for each event is called to extract useful fields from the event. *Figure 4-12* indicates the parsed mouse button and move events. It is noteworthy that `mouseEvents` and `mouseEvent2` only have `x` and `y` fields, therefore `x` and `y` are used directly as the positions of the event. However, `mouseEvent3` does not have absolute `x` and `y` but rather it has document relative and target relative `x` and `y`. The current data is insufficient to determine the accurate absolute position of events in `mouseEvent3`, therefore document relative `x` and `y` are used.

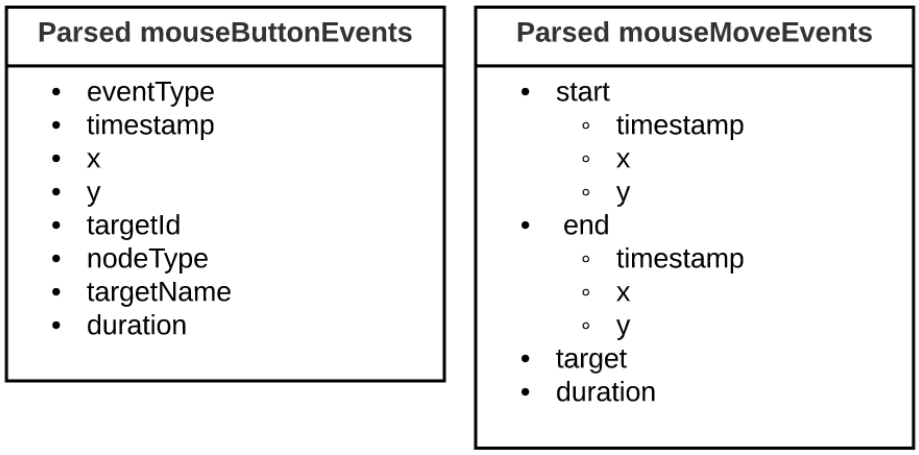


Figure 4-12 Parsed Mouse Button and Move Events

4.1.3 Screen Data Parser

Screen data consists of four parts: screen dimensions, outer(window) dimensions, inner(viewport) dimensions, and window position. In most normal cases, those data are directly applied to each corresponding component. However, there are cases when the inner dimension is larger than the outer dimension, this is potentially due to the zoom of the web page or browser. Currently the database does not include adequate information to determine the zoom level. Therefore, when any inner dimension is larger than the outer dimension, the inner dimension is applied to the outer, and the position of the window is set to zero in order to avoid a confusing screen representation.

4.1.4 Touch Event Data Parser

Touch events also have three types. *Figure 4-13* shows the different types of touch events. `touchEvents` includes touch start, move and end which the other two events do not have. Each event has a field called `touches` which points out how many touches are present at the same timestamp. For example, three elements inside the `touches` indicates there are three

points of contact with the screen at the same time.

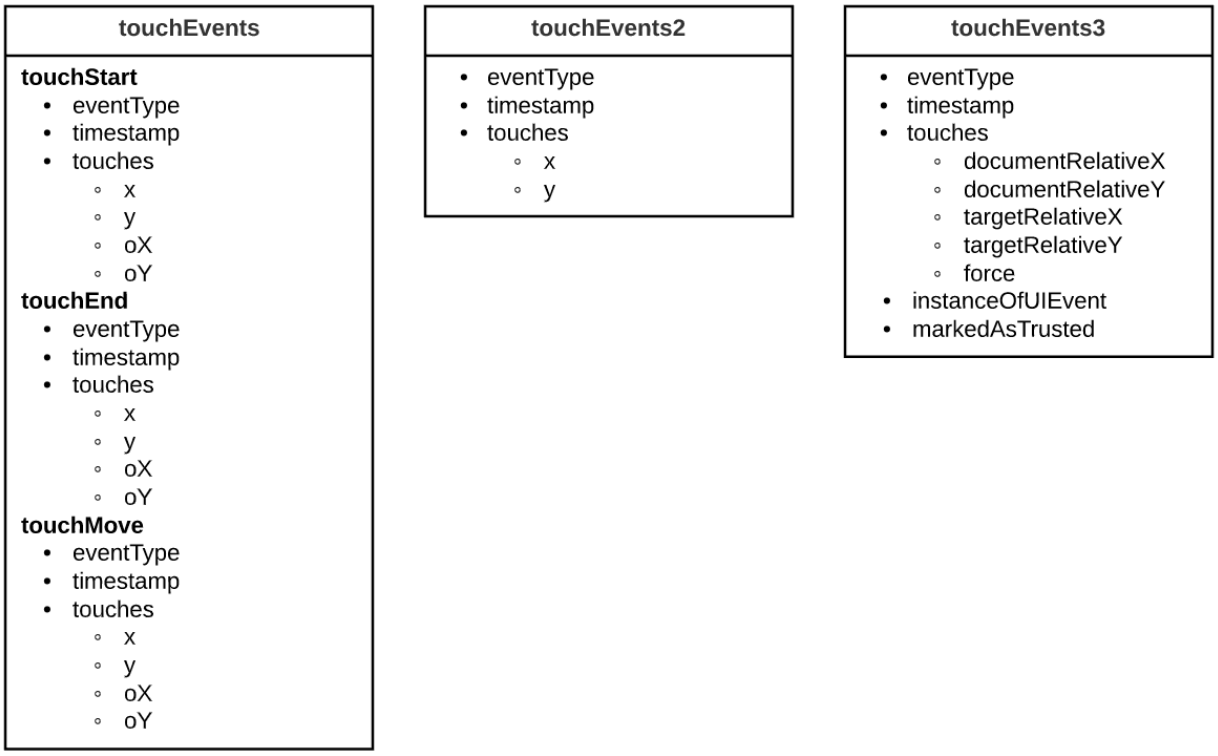


Figure 4-13 Three Types of Touch Events

Figure 4-14 shows the normalized touch event. `eventType` is an integer with 1 for touch start, 2 for touch move and 3 for touch end. `timestamp` is an integer (in milliseconds) indicating the time that event happens. *Figure 4-15* indicates the parsed touch event. A parsed touch event is created for each element inside the `touches` list. Three elements inside the `touches` results in creating three parsed touch events. This way of parsing touch events helps remove unnecessary fields and makes it easier to reproduce touch events.

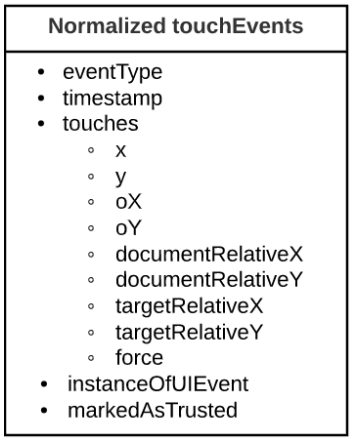


Figure 4-14 Normalized Touch Event

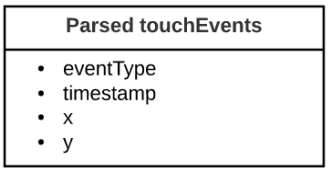


Figure 4-15 Parsed Touch Event

4.1.5 Visibility Event Parser

Visibility events are used to indicate if the user is focusing on the window or not. There are only two fields inside each visibility event: `timestamp` and `visible`. `timestamp` is an integer indicating the start time and `visible` is a boolean with true for focusing on the window and false for not focusing the window. Because visibility events only have one type, there is no need to have normalization function and the data is directly parsed.

4.2 Components Implementation

After the signal data are parsed through multiple parsers, components are created to display these parsed data. There are four main components in the application: keyboard, screen,

timeline and side panel. The following sections explain how these components are realized using ReactJS.

4.2.1 Keyboard Component Implementation

This section describes the implementation of keyboard component, the structure of the component as it appears on the screen and the manner in which it was designed to best fit into the React framework.

4.2.1.1 Component Hierarchy

Keyboard event representation consists of three components: keystroke panel, keystroke group and keystroke. This hierarchy is shown in *Figure 4-16.*

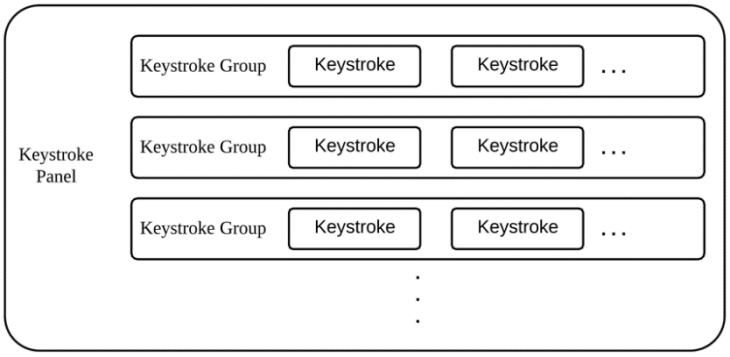


Figure 4-16 Hierarchy of Keyboard Components

4.2.1.2 Component Data Flow

The data attached to the keystroke panel component is parsed keystroke data, which is a list of keystroke groups. In React a “prop” is used to transfer the data from one parent component to its child components. When a keystroke panel component is declared inside the entry component (App.js), as shown in *Figure 4-17*, keystroke data is passed as a prop to the keystroke panel component.

```
<keystrokePanel keystrokeGroups={keystrokeGroupsData} />
```

Figure 4-17 KeystrokePanel Tag

In the keystroke panel component, as shown in *Figure 4-18*, each keystroke groups' data are separated and passed to a new keystroke group component.

```
keystrokeGroups.map((keystrokeGroup) =>
    <KeystrokeGroup keystrokes={keystrokeGroup} />
)
```

Figure 4-18 KeystrokeGroup Tag

In the keystroke group component, as shown in *Figure 4-19*, each keystrokes' data are separated and passed to a new keystroke component.

```
keystrokes.map((keystroke) =>
    <Keystroke keystroke={keystroke} />
)
```

Figure 4-19 Keystroke Tag

4.2.1.3 Keystroke Representation

Once the keystroke data passed from the upper level is received by the keystroke component. It first calculates the start position and length of the keystroke. As shown in *Figure 4-20*, the start position of the keystroke is percentage of start time of keystroke divided by the total time of the whole transaction; the length of the keystroke is the percentage of the duration of the keystroke divided by the total time of the whole transaction. Start position is passed as "margin left" to keystroke style properties and length is passed as "width" to keystroke style properties.

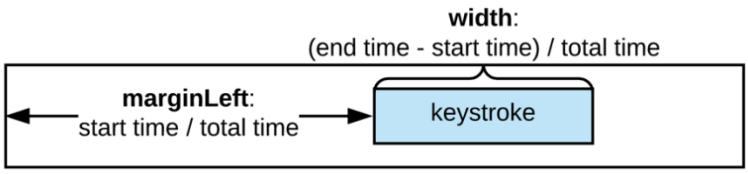


Figure 4-20 Calculation of Margin Left and Width of Keystroke

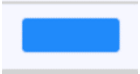


Figure 4-21 Key_down_and_up representation

Figure 4-22 Key_down representation

Figure 4-23 Key_up representation

Different key types of keystroke are represented differently. As shown in the *Figure 4-21*, if the keystroke is a “key_down_and_up” keystroke, it will be represented as a block. If the keystroke is a “key_down”, it will be represented as an inverted triangle as shown in *Figure 4-22*. If the keystroke is a “key_up” keystroke, it will be represented as a normal triangle as shown in *Figure 4-23*.

4.2.2 Screen Component Implementation

Screen component in VizMonkey 2 is used to reproduce the screen, window, viewport and user interaction events of a transaction. This section discusses the hierarchy and implementation of each component inside screen part.

4.2.2.1 Component Hierarchy

The screen component contains three major components: the screen component, window component and viewport component. The hierarchy of the above components is shown in *Figure*

4-24. Each component is rendered by using its own width and height.

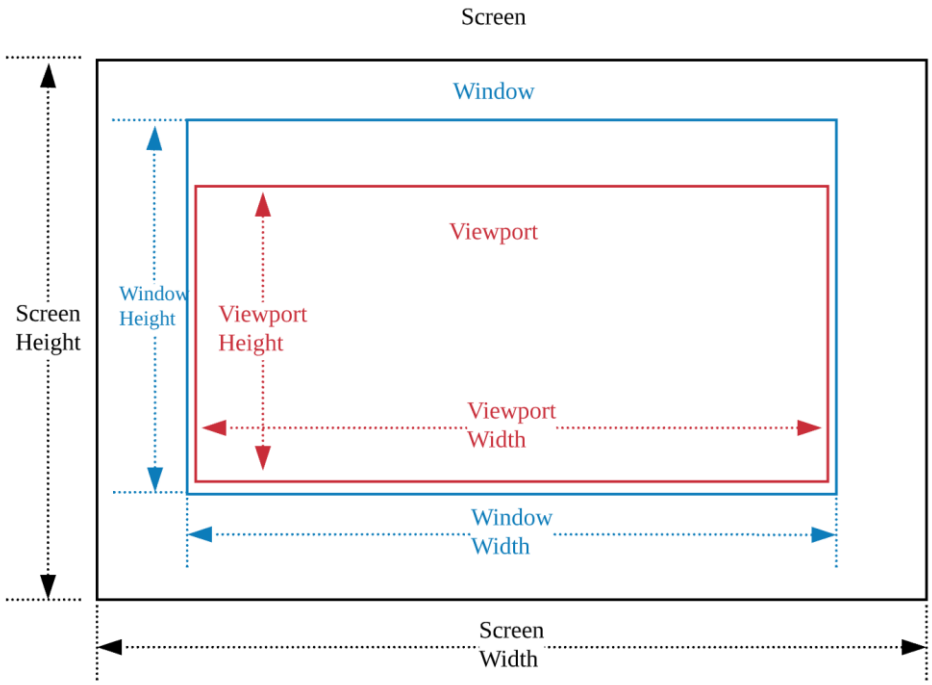


Figure 4-24 Structure of Screen Part

4.2.2.2 Screen Component

Screen is the user's actual monitor or desktop size. HTML basic tag `<div>` is used to reproduce the screen. Once the data is loaded from BigQuery, the root component, App.js, dispatches an action which sends screen related data to the screen reducer. When the screen reducer receives the data, the width and height of screen dimensions from database are divided by the width and height of browser from the VizMonkey 2 user's screen. This results in two different ratios: width ratio and height ratio. To achieve the maximum filling of the display area without changing the height and width ratio of the screen. The ratio used is the one which is the most constrained by the user's browser. This ratio is also be used to scale all the subcomponents of the screen. The screen component is connected to the screen reducer and is automatically re-rendered when the data in the screen reducer change. For example, when the VizMonkey 2 users

change their browser's dimensions, the screen scaling ratio is dynamically recalculated, resulting in re-rendering the screen component.

4.2.2.3 Window Component

Window refers to the user's browser window, including the UI, the menu bar and the viewport inside it. Within the screen, the accurate dimension and position of the window is reproduced by using HTML tag `<div>`. Then, the dimension and position are both scaled together by the screen scaling ratio to maintain an accurate representation of the window relative to screen. A URL bar is reproduced on top of this window. Due to the limited data collected, it is unknown if the console is open. A console window opened on the side results in a difference in width between window and viewport, showing a blank area on the side to be considered as a console. However, since the height of the header is unknown, the difference in height between window and viewport contains the header and any opened console on the bottom. Currently, the VizMonky 2 limits the maximum header height at 35 pixels, and the remaining difference appears as a blank area on the bottom to be interpreted as an opened console. *Figure 4-25* shows a situation in which the header of the browser window is larger than 35 pixels.

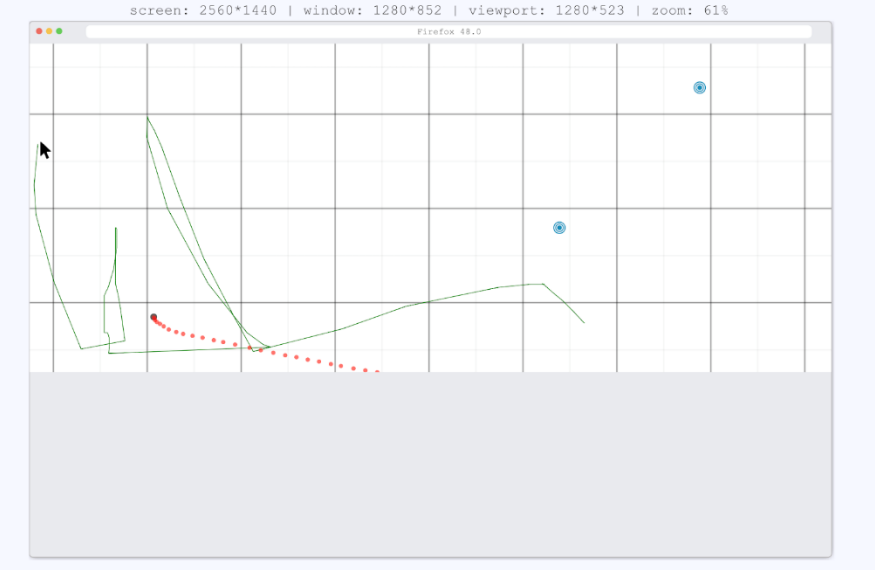


Figure 4-25 Example of Window with Opened Console

Visibility events are rendered inside the window component by putting a grey transparent overlay above the window as shown in *Figure 4-26.* When the visibility event is true, CSS property visibility is sent to visible to show the overlay and when false, visibility is set to none to hide the overlay.

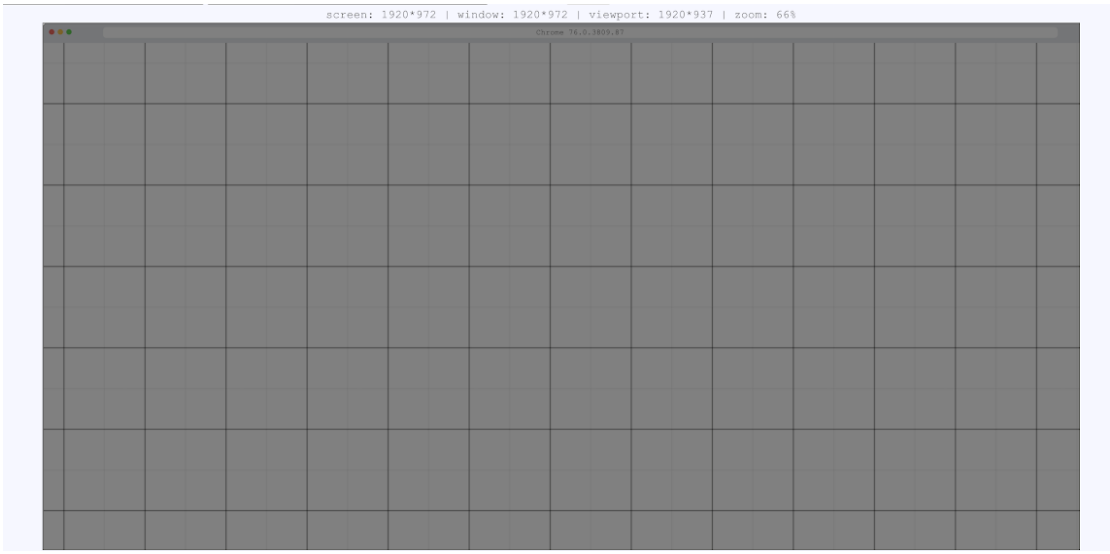
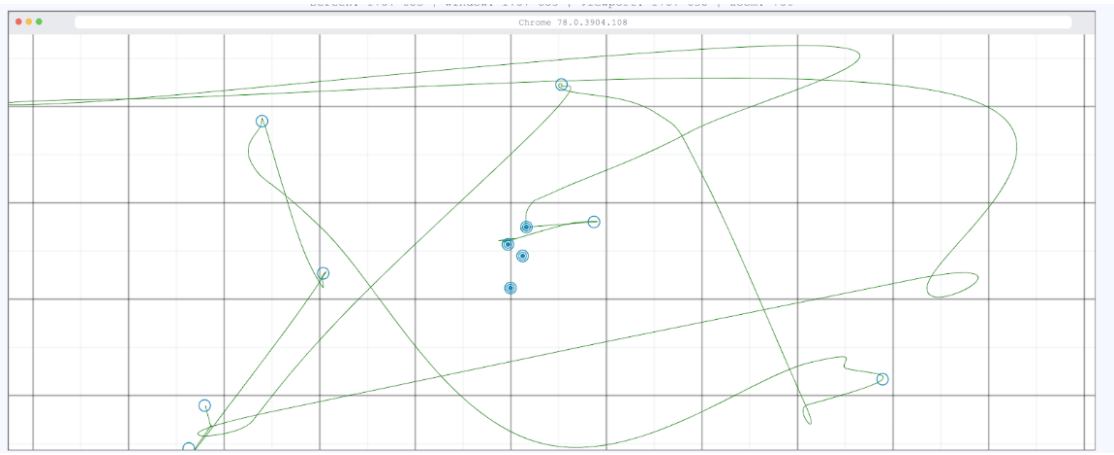


Figure 4-26 Example of Visibility Representation

4.2.2.4 Viewport Component

The viewport refers to the user's visual area of a web page. The background of the viewport is filled with grid lines which indicate the resolution of the web page. A denser grid indicates a higher resolution. Inside the viewport, all user interaction events that have locations are reproduced using SVG elements. The most common event is the mouse movement. For each of the mouse movements, a `<path>` element is created. The `<path>` element has an attribute called d which is used to specify the shape of the path element. The start and end points of a single path are passed to the "d" attribute to link the two points. However, the sample rate of the original mouse data is not high enough. The low fidelity results in straight lines and sharp corners which look unnatural. In order to better mimic the mouse movement of the user, the Bezier curve algorithm is used to smooth the paths and produce lines in a more reasonable curved shape. *Figure 4-27* shows comparison of non-interpreted curve and Bezier curve.



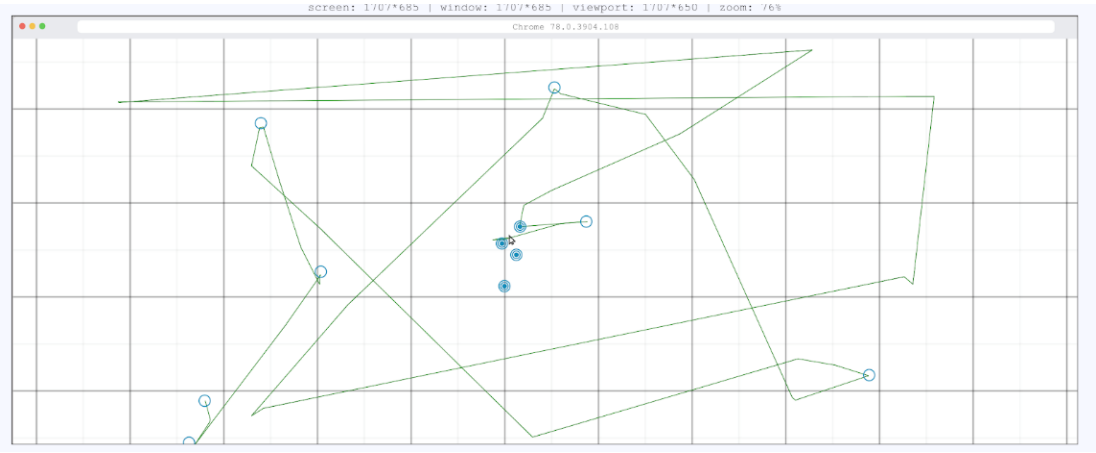


Figure 4-27 Comparison of Normal Curve and Bezier Curve

Mouse clicks are drawn by using the `<circle>` element. Different sizes of circles are used to represent different types of button events. The largest hollow circle represents mouse down, the middle hollow circle is mouse up and smallest solid circle is mouse click. In addition, different types of clicks have different colors. When the user hovers over each button event, the event is highlighted in red and a text box pops up to show the target ID and the click type of the current event. *Figure 4-28* shows an example of this type of pop up text box.

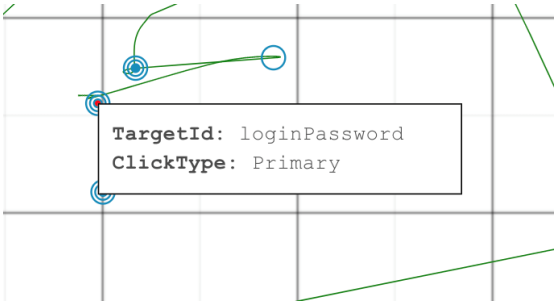


Figure 4-28 Example of Pop Up Box

A touch event is also represented by a `<circle>` element. `touchstart` and `touchend` are displayed with larger circle to highlight their difference from the rest of the data points. One touch event usually consists of many `touchmove` events in-between `touchstart` and `touchend`. In the case of multiple touches, the data was collected in one

group; this makes it impossible to differentiate different fingers. On the other hand, the touch events data are collected in high frequency and density which allows VizMonkey 2 users to differentiate each cluster of touch data during visual inspection. *Figure 4-29* shows an example of touch events.

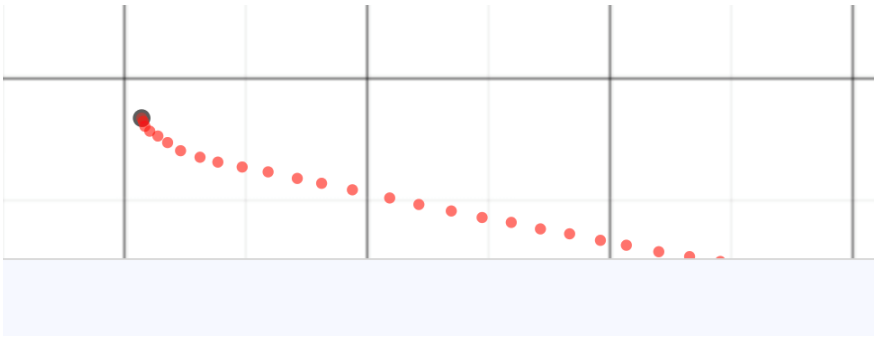


Figure 4-29 Example of Touch Event Representation

4.2.3 Timeline Component Implementation

Timeline in Vizmonkey 2 is an object created by AnimeJS. This section describes some fundamental concepts about timeline in AnimeJS and the data flow within the timeline component.

4.2.3.1 AnimeJS Timeline

In order to control the animations by simply scrubbing the slider, a timeline object is created to interact with the slider. *Figure 4-30* shows the procedure to declare a timeline object in AnimeJS.

```
const tl = anime.timeline({
  loop: false,
  autoplay: false,
  duration: 800
});
```

Figure 4-30 Declaration of Timeline Object

"Loop" decides whether or not the timeline object should loop the animations inside it. "Autoplay" decides whether the animations should be automatically played without pressing the play/pause button. "Duration" represents the total time duration of all the animations in the timeline.

After the timeline object is created, animations can then be putted onto the timeline. *Figure 4-31* shows the way to declare an animation object in AnimeJS.

```
const animation = anime({
  targets: '#square',
  scale: 2,
  translateX: '200px',
  duration: 2000,
});
```

Figure 4-31 Declaration of Animation Object

Target, specific animation transition, and animation duration are needed to create the animation object. The target represents the HTML DOM object to be manipulated. Class name, id, HTML tag name can be put here as the specifier. For animation transition, there are multiple fields that can be animated by AnimeJS. For instance, the size property, the position property, or background color property. The duration represents how long this animation should last.

4.2.3.2 Data Flow

Since the timeline object should be accessed globally, the best place to declare this object is within the entry point of the application, which is App.js. After the timeline object is created, it is passed to the Timeline component (with all the parsed data) to initialize all the animations that the timeline object needed.

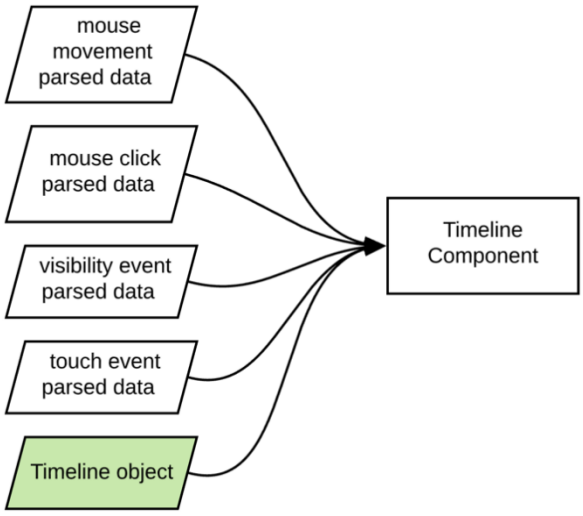


Figure 4-32 Data Flow from App to Timeline Component

As shown in *Figure 4-33*, in App.js, mouse movement, mouse click, visibility, and touch data are all sent to timeline component along with the timeline object after they are parsed.

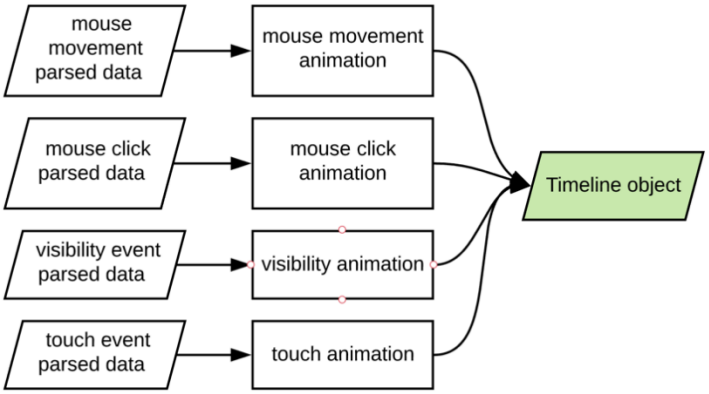


Figure 4-33 Timeline Object Construction

Once the timeline component receives all the parsed data along with the timeline object, as shown in *Figure 4-33*, parsed data are converted to their corresponding animations and then the timeline object puts the animations into itself.

4.2.4 Side Panel Component Implementation

The side panel in Vizmonkey 2 is used to display basic information of the transaction being played along with the transaction log. This section discusses the component hierarchy of

the side panel component, and then describes the two child components within the side panel component.

4.2.4.1 Component Hierarchy

The side panel component consists of two components: the session info panel component and the events list panel component. The session info displays some basic information about the current transaction including, but not limited to, the transaction id, transaction date, and user agent. The events list panel contains all the events in the transaction. *Figure 4-34* shows the component hierarchy of the side panel.

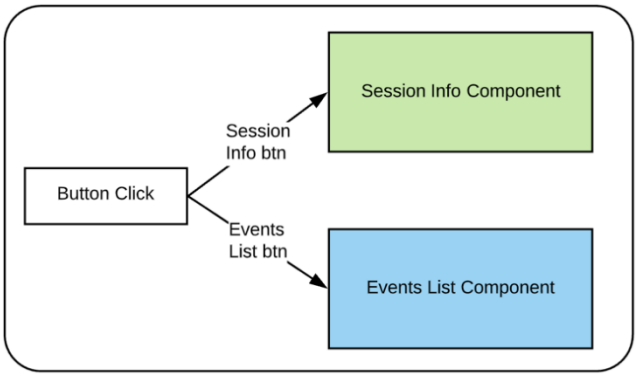


Figure 4-34 Component Hierarchy of Side Panel

At the start of the application, the session info is rendered above the events list. The only difference is that in the style sheet, the session info component is visible, but the events list component is not visible. The button in the side panel is used to switch the CSS style of the two components as shown in *Figure 4-35*. The CSS style in the session info and events list component is changed when the button is clicked.

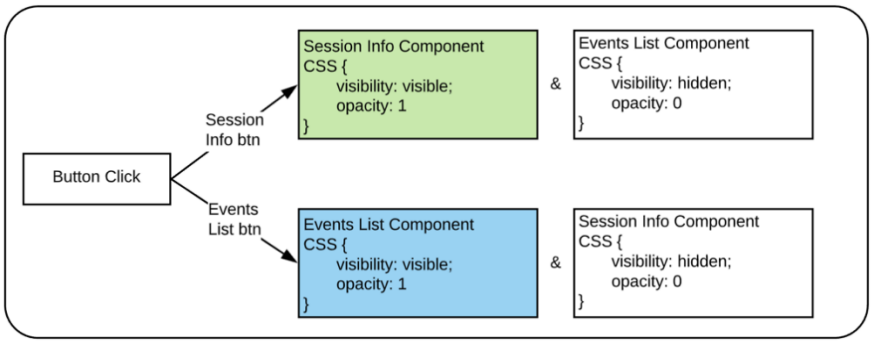


Figure 4-35 CSS Transition of Side Panel

4.2.4.2 Session Info Panel Component

As shown in *Figure 4-36*, the session info panel displays various types of basic transaction information such as transaction ID, transaction date, IP address, browser type, operating system, and website. These data are from "propertyValues" in the signal data. Conditional rendering is used here to display the icon corresponding to the operating system and browser type.

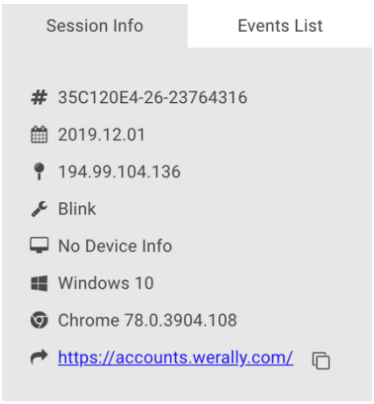


Figure 4-36 Session Info Example

4.2.4.3 Events List Panel Component

The hierarchy of the events list panel component is shown in *Figure 4-37*. The events tab shows detailed information of the events from the transaction. For instance, the mouse movement event shows the start and end positions of the movement, the mouse click event

shows the type of click (mouse down, mouse press or mouse up) along with the target of the mouse click, and the keyboard event shows which key is pressed and where the user types the key.

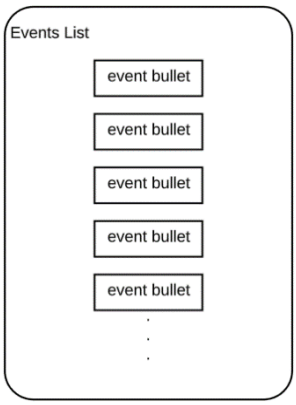


Figure 4-37 Hierarchy of Event List Component

The events list panel also has the ability to interact with the timeline. If the transaction is currently playing, all past events in the panel are highlighted. *Figure 4-35* shows the life cycle of the events list component and timeline component.

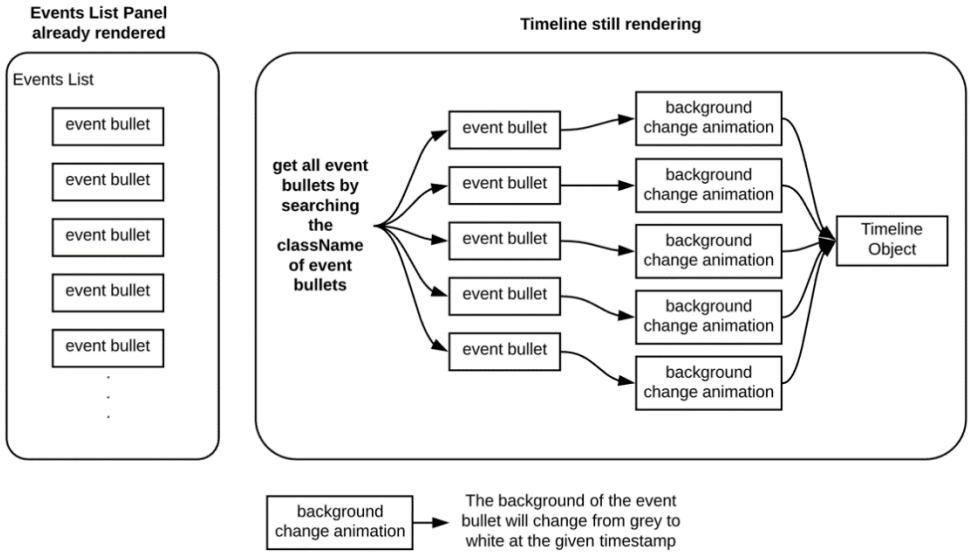


Figure 4-38 Relationship Between Events List Panel and Timeline

As shown in the *Figure 4-38*, after all the events in the events list panel are rendered, the timeline component searches for all the events bullets and makes background-change animations. After the animations are generated, they are put into the timeline.

The events list panel also has a feature that auto-scrolls to the latest event. In order to auto-scroll, certain event data must be available. *Figure 4-39* shows the data necessary for the calculations.

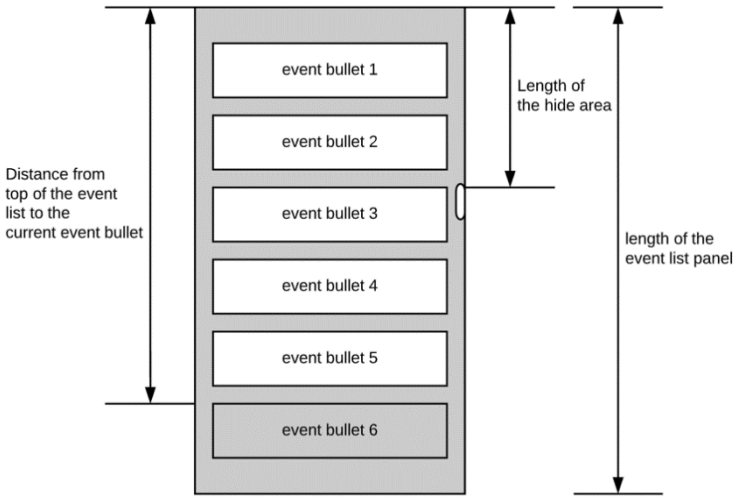


Figure 4-39 Auto-scroll Calculation

The data available via the built-in JavaScript are the distance from the top of the event list, the length of the hidden area (the area hidden when the scroll bar is scrolled) and the physical length of the event list panel. After all these data are acquired, the calculation is:

Distance from top of the event list to the current event bullet - Length of hide area + event bullet height

Figure 4-40 Auto-scroll Formula

If the numbers from the above calculation are larger than the length of the event list panel, then the scroll bar needs to be scrolled by the height of the current event bullet.

5 Evaluation

During the creation of Vizmonkey 2 as well as after it was completed, we tested its technical robustness through automated testing and its effectiveness by showing it to its end users and asking for their feedback. The user feedback was mostly positive, and the automated testing reveal no bugs which were left unfixed.

5.1 User Responses

In order to gauge user responses to VizMonkey 2 we showed it to a crowd of about 10 possible users and asked them to comment on certain interactions. They were very positive towards the keystroke abstractions; they appreciated the added ability to look at a transaction statically. They were also very pleased with the overall cleanness of the UI. Their favorite feature was the ability to scrub through transactions and go to specific points. They had two main criticisms about the version of VizMonkey 2 we showed them. Firstly, they did not like how you had to hover over an event in order to see its contents. Secondly, they were worried that the realistic screen representation would not be able to display odd resolutions. They were otherwise satisfied.

We addressed these concerns by warning the user if there was a resolution which was inaccurate and by adding in the event list into the final version of the application. When the users saw this final version, they had no concerns and only had requests for new features.

5.2 Test Suite Development Process

Our automated tests were created in both Jest and Enzyme. Jest is a JavaScript unit testing framework especially well suited to testing React components. Enzyme is a testing

framework that has additional functionalities to test React components, specifically their dynamic properties and animations. For VizMonkey 2, Jest was used as the test runner, mocking library, and assertion library and Enzyme was used to provide utilities to render components, find components, and interact with elements.

Jest has a method called `describe` which takes two arguments: a string for explaining the test and a callback function. `describe` will group related tests together into one block. Another method is called `it` which takes two arguments: a string for test name and a function that contains test expectations. Actual tests usually exist in the second argument of `it`. Figure 5-1 shows an example of using `describe` and `it`.

```
describe("React component", () => {
    it("should render its first child ", () => {
        //test
    })
    it("should render its second child", () => {
        //test
    })
})
```

Figure 5-1 Jest Example Code

To find the components we want to test, the custom attribute `data-test` is used. There are other attributes that can be used to find the target components such as `className` or `id`. However, those attributes are easily changed by programmers while making normal changes to the application. To avoid rewriting tests when `className` or `id` changes, a custom attribute `data-test` is created for testing.

Tests for ViMonkey 2 include three major parts: tests for rendering the components, tests for functionality, and tests for reducers and actions. The Enzyme methods `shallow` and `mount` are used to render components. `shallow` is used to render single component and it does not

render its child components. `mount` is used to render whole component including child components, but it has a higher cost than `shallow`. To test whether components are successfully rendered, `shallow` method is used more often in our testing process to increase testing speed.

In order to test functionalities of components, mock function will be created by calling `jest.fn()`. For example, to test whether onClick event on a button is correctly handled. Clicking the button triggers a function and this function can be triggered by using `jest.fn()`. Then we can to see if the callback function is successfully called by that trigger. Most of the functions within VizMonkey 2 are tested using mock functions as they would normally be triggered by user interaction. Some of these functions need parameters. Parameters are also supported by Jest's trigger functions.

VizMonkey 2 used redux to help manage data flow. Therefore, we needed to test the reducers and action creators. To do this a fake store is setup then we create different situations of states. Once these simulated states are created, we pass the states to target reducer to see if the output state is the same as expected.

6 Conclusion

VizMonkey 1 was a data visualization tool which Shape had already implemented. VizMonkey 1 utilized Shape Client data to create representations of user interactions so that shape employees could look at individual interactions and see why a certain one was labeled as automated or not and give this explanation to customers. The data visualizations included mouse and keyboard events.

As VizMonkey 1 became more utilized within the company its limitations became an increasing hinderance. Its representations, especially of keyboard events, were difficult to understand without prior explanation. It showed keyboard events as completely separate regardless of the key of origin; this made it impossible to see the duration of keystrokes. Mouse events were also hard to follow as mouse locations were not connected; the locations where shown as static dots. Intermediate locations were not displayed.

Shape identified many problems with VizMonkey 1 and asked that we address them by making a new version. It gave us this list of VizMonkey 1 Shortcomings:

- Keyboard and mouse information is hard to understand
- Not all data is represented
- The animation requires explanation and training
- There are no playback controls for interaction with the application

Our goal was to design and implement VizMonkey 2.

We designed the user experience (UX) rather than starting from the data we were using. Starting from an abstracted point of view required more manipulation of the data and created unique problems, such as what to do with impossible screen dimensions. Once our rough design was finalized, we created the application in React. In order to allow for data flow and

responsiveness we also used Redux. Lastly, we used the AnimeJS animation framework to allow for a unified timeline.

We created representations of events which were more easily understood by non-experts and more quickly understood by experts. The use of the keystroke abstraction, in particular, was well-received by the Shape Intelligence Center's (SIC) team. A UX centric design for data visualization in this context was very successful and better than the previous data centric visualization. VizMonkey 2 is currently deployed and accessible to the SIC team and meets or exceeds the usability and functionality of VizMonkey 1.

7 Future Work

VizMonkey 2 could be improved with several features desired by Shape, such as a deeper integration with current Shape Intelligence Center's tools, the ability to show multiple transactions at once as a heat map, added ability to show phone events such as orientation in space, and using feedback from VizMonkey to improve the data Shape Client collects.

Deeper integration with SIC tools would allow faster access to a specific transaction. Rather than a user finding the database and transaction ID of the transaction they were trying to input, the user could simply click a button within another tool where they were viewing a transaction and it would allow them to see that transaction within VizMonkey. This feature would be easy to implement and save users' time.

Showing multiple transactions at once would allow for users to make comparative assessments about transactions. These comparisons would be useful in determining whether a transaction looks like it came from a bot; a bot's actions may look like those of a human, but if it is identical to many other transactions then that bot is likely a recording—there are also other bot behaviors which look less suspicious in a single transaction. In order to show multiple transactions together, a unified list of transactions between the events would need to be created and then played back at once as if it were a single transaction; this would require a reworking of the keyboard events as they are not designed to be overlaid on top of each other. Making keystrokes translucent could fix this problem. Mouse strokes would work without modification under this system by simply placing them on top of each other as their backgrounds are transparent. Once this feature is implemented comparisons between transactions, especially groups of transactions, will be much faster.

To add orientation within space, a CSS transform could be performed on the finished screen representation. The CSS transformation could be made using accelerometer data collected by Shape Client. The menu and representation of the screen should change when looking at a smartphone event. They should not include a window as smartphones do not use a windowed environment. This should also be reflected in the available menu options when viewing a smartphone transaction.

Improving Shape Client with the knowledge gained from making VizMonkey 2 would allow many more features in future versions of VizMonkey. One of the most evident features would be the ability to overlay the website as the user was seeing it. This feature only requires a few more data points, specifically the location of the document relative to the viewport. The addition of this feature was requested by several employees but is simply not possible with the current data provided by Shape Client.

References

1. “ReactJS.org” [Online]. Available: https://reactjs.org/. [Accessed: 12-March-2019]

2. “Redux.js.org” [Online]. Available: https://redux.js.org/. [Accessed: 12-March-2019]

3. “anime.js.” [Online]. Available: https://animejs.com/. [Accessed: 10-Dec-2019].

4. “W3C,” W3C SVG Working Group. [Online]. Available: https://www.w3.org/Graphics/SVG/. [Accessed: 11-Dec-2019].