Dataless Sharing of Interactive Visualization

Mohammad Raji, Jeremiah Duncan, Tanner Hobson, and Jian Huang

Abstract— Interactive visualization has become a powerful insight-revealing medium. However, the close dependency of interactive visualization on its data inhibits its shareability. Users have to choose between the two extremes of (i) sharing non-interactive dataless formats such as images and videos, or (ii) giving access to their data and software to others with no control over how the data will be used. In this work, we fill the gap between the two extremes and present a new system, called Loom. Loom captures interactive visualizations as standalone *dataless* objects. Users can interact with Loom objects as if they still have the original software and data that created those visualizations. Yet, Loom objects are completely independent and can therefore be shared online without requiring the data or the visualization software. Loom objects are efficient to store and use, and provide privacy preserving mechanisms. We demonstrate Loom's efficacy with examples of scientific visualization using Paraview, information visualization using Tableau, and journalistic visualization from New York Times.

Index Terms—shareable visualizations, visualization recordings, user interaction

1 Introduction

INTERACTIVE visualization is crucial when people need to answer questions with data. Its power comes from the interactions and visual responses that continuously refine the questions we ask. Along with the accelerating growth of "big data", interactive visualizations are also increasingly used as a shared medium by large and collaborative teams [1], [2], [3]. As such, methods to share interactive visualization are becoming more critical.

However, interactive visualization has a close dependency on the data and the software system that creates and runs the visualization. As a result, to a vast majority of everyday users, sharing interactive visualizations can become overwhelmingly expensive because of: 1) the size of the data, 2) the platform the visualization software runs on, and 3) the rules and regulations that govern and constrain the export of the data.

Currently, the cheapest way to share visualizations is to use traditional methods, such as static images and video recordings. These traditional methods lose interactivity and diminish the analytical value of the shared visualization. However, it's worth noting that these old methods provide dataless sharing of scripted visualizations. "Dataless" in the sense that the original dataset is no longer required by a subsequent user, and hence the shared visualizations can be free from any of the above cost barriers.

In this paper, we present a method for dataless sharing of *interactive* visualization, where a shared visualization no longer relies on its original data source nor the original software, yet a fully customizable amount of interactivity is maintained and controlled by the creator of the shared interactive visualizations. The aspect of sharing as well as maintaining interactivity entails the following special design considerations.

Efficiency. Traditional interactive visualizations depend on their datasets. That dependency is natural and inherent, however, it can be expensive. If a dataset is larger than affordable to store, process, or send over a network, it becomes infeasible to share with others. In contrast, traditional dataless visualizations (e.g. images and videos) occupy much less space and require very modest computing power, making them easy to share. Therefore, it is important that our approach be efficient in both storage and computing needs when adding interactivity into the mix.

Sustainability. When an interactive visualization is used as a tool, the implied life duration is very short. When shared, the duration has to be much longer, if not eternal. Web-based visualizations have taken a good step towards this. However, as technology evolves, better tools will replace older ones, and older visualizations become harder to reproduce. An example of this are the many visualizations created using Adobe Flash that are now obsolete. Without any data or software dependency, images and videos are very sustainable. We wonder if we can find a way to do the same for dataless interactive visualizations and provide a general and reproducible format, and study what costs this entails.

Controllability. Traditionally, when one has the software and data for a visualization, they have total access. To share that visualization, they need to share that full access with recipients unless they substitute the visualization with non-interactive dataless alternatives. Many datasets are never shared in forms beyond non-interactive images or videos due to this reason and as a result of organizational policies. Therefore, it is beneficial to be able to create interactive recordings with variable access policies depending on the audience and regulations that govern the data [4].

With regard to these requirements, we have developed an approach to record visualizations along with controllable degrees of interactivity. The prototype system is called *Loom*. Loom can capture interactive visualizations as dataless independent objects. During creation, a user (video creator) sets the variety of user interactions that are allowed by recipients. Loom then provides an interactive recording of the visualization. Loom recordings are viewable in web-browsers without the need of any plugins.

For instance, a Loom recording can capture a set of visualizations in the Tableau application. The recording becomes completely independent from Tableau and any data or server connections that retrieve data products. The recording can then be interacted with offline, shared, as well as viewed online without Tableau installed. As another example, interactive recordings of a volume rendering technique in Paraview or VisIt can be submitted along with a paper submission without exposing their data or source code. Later the same recording can be shared on the web so that the audience that reads the work can also interact with different areas of the

• *The authors are with the University of Tennessee, Knoxville.*

rendering online.

While Loom focuses on shareability, it is also related to the topic of visualization archiving. In particular, we note that "archiving for the purpose of sharing" is different from "sharing with a possibility of archiving", and we feel the scope of Loom is more modestly the latter. This is because scientific archives have two missions: (i) to make results openly available in perpetuity, and (ii) to organize scientific information in a narrated way that allows search and empowers open scientific inquiry [5]. To this end, this paper aims at sharing interactive visualizations openly, in perpetuity, with a possibility of archiving, and without needing to preserve the original software nor the data used to create those visualizations.

Note that while Loom recordings can include exploratory visualizations that include many application states, they are mainly created for explanatory visualizations [6] in which the interactive workflow is pre-determined and known. Additionally, similar to video recordings, Loom recordings cannot capture the entirety of the exponentially large interactive space of a visualization. Instead, they make efficient shareability of the interaction experience possible and fill the gap in the shareable-visualization spectrum.

To capture the interaction space of a visualization, Loom models user interactions as an action tree. Given an interactive visualization tool, a creator can use Loom's Overlay Application (LOA) to specify different UI areas and indicate to Loom which interactions those UI areas support (e.g. clicking, brushing). Then, LOA uses OS-level UI automation to interact with the visualizations and capture its different states as images. It then organizes and compresses these images into a single object. Essentially, our methodology takes a black-box perspective with respect to interactive visualization where one can consider an interactive visualization as a function, which accepts user actions as input and provides visual responses, as output. While this is a very simplified view, it enables modeling and reproducing interactive visualization experiences of widely varied applications. The Loom project will be made open-source.

In summary, our contributions are the following.

- Presenting a capture/reconstruct method for the recording and sharing of dataless interactive visualizations, making them independent of the original data and application source code.
- Enabling the control of access policies on top of existing interactive visualizations.
- A prototype implementation of our method called Loom with a smart toolset that assists users in capturing interactive visualizations.

The rest of this paper is as follows. We discuss the background of our work in Section 2. The methodology of our system is explained in Section 3. Section 4 covers three types of applications (desktop-based information-visualization, web-based information visualization, and scientific visualization) captured using Loom. Results and a discussion on the limitations of Loom are discussed in Section 5. The paper is concluded in Section 6.

2 Background

2.1 Shareability and Provenance

Visualizations are often shared to communicate insights. Traditionally, shared visualizations have taken the form of non-interactive images and videos, or otherwise niche-focused and custom-built applications [7], [8]. Modern web applications have taken a great step towards general shareability of interactive visualization with the advent of flexible libraries [9], [10] and Javascript frameworks with data-binding support [11], [12]. For example, the species-mapper tool from the National Park service provides the option of sharing the state of the visualization with a hyperlink that reproduces the state when navigated to [13].

However web solutions either require dedicated data servers or small data sizes that can be transferred to the client. Additionally, these approaches make the data and software accessible to all of their users on the web. Also, software used for visualization can become obsolete as is evident by the discontinuation of Adobe Flash [14], and data ownership policies mean that data sources may not always be available, therefore hindering shareability.

Sharing visualizations is not limited to the visuals themselves, but also the process of analysis through the topic of provenance. Heer at al. consider visual analysis as not just the generation of visualizations statically, but as an iterative process that is refined through interaction [15]. As such, they denote recording and sharing this process as crucial components of the visual analysis taxonomy.

VisTrails [16] systematically captures provenance information in a tree structure and is used in conjunction with other visualization tools to store and recreate workflows. In Paraview [17], Lookmarks have been used to store and share views of datasets similar to how bookmarks work for webpages [18].

The topic of provenance has also been looked at in the context of web applications. The Open Provenance Vision [19] has been presented as a general vocabulary and format that can be used by different semantic web applications. As follow-up work, the W3C presented the PROV standard as a set of specifications for storing and sharing transformations to data [20]. Many applications have been built on top of such web-based specifications, such as Komadu [21], and Karma [22].

Within web-based visualizations, SIMProv.js has been recently introduced as a general way of augmenting web-based applications to include provenance throughout the user's interactions and reasoning process [23].

While Loom recordings can be used to communicate a certain workflow or a user's analytic process, our aim is to allow users to have a sense of flexibility in their interactions. Additionally, Loom is not dependent on a particular visualization type or application and supports a variety of visualizations.

One of the most related works to Loom is Graphical Histories [24], in which the states of a visualization application are stored as a hierarchy of images depending on user interactions. Similar works exist that store image transformations in hierarchical structures [25], [26]. Loom takes this idea further and recreates the interactivity of the application at runtime so that the visualization can be used while the data and code no longer need to exist.

2.2 Automation in Visualizations

Given the highly visual and interactive aspects of visualizations, a Graphical User Interface (GUI) of some sort is often assumed with visualizations. When using these GUIs, a common user behavior is to search through a problem space in search of significant patterns.

As common in computer science, the search can be automated even for visualization. For example, automated compound boolean query based visualizations [27], automated regular expression based queries [28], even one of the most difficult tasks in visualization, the design of effective transfer functions [29], [30],

[31]. These automations employ simple, powerful visualization specific languages, and as a result, have led to many visualization researchers considering the specification of visualization as textual.

In the above cases, researchers used special program-accessible interfaces to control the visualizations and found great successes. In a related way, other researchers have also built and used UI-bots to automatically go through a graphical user interface. One of the most recent works is the use of monkey testing to automatically stress-test web based visualizations [32], [33].

In this work, we have also developed a UI-bot and ways for a human expert (e.g. a visualization developer) to guide the UI-bot to methodically go through the problem space as specified by the graphical user interfaces. Our key focus here is that, in a transparent way, the captured visualizations are organized according to guidance provided by the human expert.

2.3 Image and Video Compression

Storing the captured raster images can take a lot of space and a small size is vital to shareability. Therefore, appropriate compression is required. Visualization researchers have always been dedicated users of the most advanced image and video compression techniques, such as JPEG, PNG, H.264 etc.

Image compression has been used extensively whenever a screen capture or a framebuffer has been exported as the end result of a visualization. However, image-based visualization systems have also significantly leveraged image compression to store intermediate visualization products. Some of the most innovative systems, such as Cinema [34] have built advanced systems around image-compression technology so that specialized visualization objects can be efficiently compressed, managed and creatively re-used. To this end, image-compression has been used not only as end products of visualization, but also as intermediate products of visualization.

Compressing visualizations as videos is common. However, its usage has mostly been limited to when a visualization is a time-sequence treated as a video. Sometimes, in a remote visualization setting, videos are created live when a visualization is being computed interactively [35]. To this end, videos are primarily end products of visualization.

From this respect, to our knowledge, our work is one of the first to develop specialized ways to leverage video compression technology to store images as intermediate products, so that users can later make unconstrained use of visualizations.

Video compression techniques often leverage the similarity of consecutive images that are stored in them as frames. Our work benefits from this fact, since our captured images follow a natural order that maximizes the similarity between consecutive frames. Additionally, video decompression is automatically supported and is an optimized feature in browsers. Loom relies on the fast in-browser decompression of videos at runtime.

2.4 Capturing and Modeling Application Behavior

Interaction with applications and managing their state have been modeled with UMLs and other types of finite state machines for many years [36], [37]. UMLs provide a complex state machine that can represent how one can interact with a user interface and how the state of the application changes with actions. However, UMLs can quickly become very large and complex. In recent years, behavior trees have been used to model artificial intelligence in games [38] as simpler and more modular alternatives. Originally, behavior trees were designed by Dromey et al. as a way to formalize requirements in designing systems [39].

Inspired by behavior trees, we use a tree structure as a simple way to model interactions in a visualization application. When using Loom objects at runtime, our system converts the tree to a state machine in order to partially reconstruct the original visualization's behavior.

Capturing and storing an entire application along with its data and code has been possible for a long time with the help of virtual containers. Containers can be used to store applications along with their data and requirements [40]. While containers simplify installation and reuse, the size of the resulting container images can grow exponentially. Additionally, any system that is not based on processed products of the application requires the presence of the data source or at least a connection to it. However, this is infeasible when one wishes to not share their data or source code (e.g. in paper submissions). This is also difficult when the size of the data increases, making web-based sharing infeasible.

3 Method

To support recording the large variety of visualizations and platforms that exist today, some of which can be offline, Loom takes a platform-agnostic and black box perspective on interactive visualizations. In this view, the input to visualization applications are user actions, and the output are visual responses. Loom then models this input/output behavior.

As with user interfaces in general, interactions with a visualization have a hierarchical nature, in that with every action, a new set of options to interact with are given to the user. Inspired by behavior trees [38], Loom models the hierarchy of actions in a visualization using an *action tree* (Section 3.1). It then traverses the tree and takes those actions automatically to capture the visual responses associated with each action. The actions and the responses are then stored together as a Loom recording (Figure 1-top). This is detailed in Section 3.2.

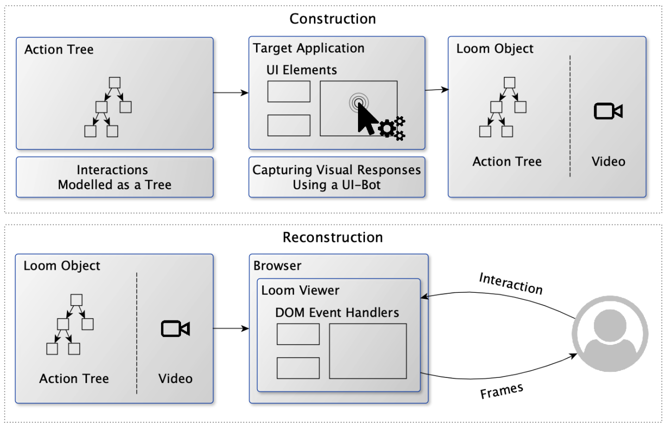


Fig. 1: The overview of the Loom system. The two stages of a Loom object's lifecycle are shown. The interactions with a visualization are modeled using an "action tree". The tree is then traversed to capture visual responses as a compressed video. The tree and video are then used to reconstruct the interactive visualization in a browser.

At runtime, to view and interact with the recording, our browser-based viewer opens the Loom recording and reconstructs the visualization application using HTML's DOM event handlers (Section 3.3). Based on the user's interactions in the browser, appropriate visual responses are picked by Loom and shown (Figure 1-bottom). To the end-user, the experience is as if they are interacting with the original visualization and data.

Specifying the action tree is done by the creator of the recording. Loom provides a rich toolset that assists the creator in action specifications (Section 3.4). Loom's supported interactions can be extended using plugins (Section 3.5). The last section of the our methodology describes the control of privacy using encryption.

3.1 Modeling Interactive Visualizations

Every visualization workflow consists of a set of interaction sequences that users perform. Consider the Tableau visualizations in Figure 2. These interaction sequences could be as simple as: click on the "Overview" button to launch the overview window, and then mouse over individual states on the map for more information. Another sequence could be to click on the "Profit Ratio by City" button, and then scroll through the bar graph in the new window. While some interactions provide more flexibility than others (e.g. brushing vs. clicking), with every interaction, users are presented with further options that can be picked and interacted with.

Each sequence can form a different hierarchy of possible float of actions for users. Loom models these options as *action tree*. As with behavior trees, action trees express the various states of an application and the flow of control from one state to another. A simplified version of the action tree of the Tableau example is shown in Figure 3.

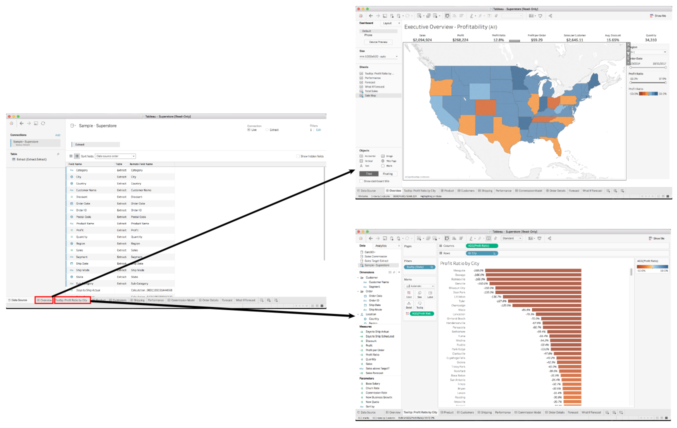


Fig. 2: The Tableau application (left) showing the sample superstore dataset. Clicking on the first highlighted button at the bottom of the screen (Overview) shows a map view. The next button shows a bar graph of profit ratio by city. It is only after clicking the Overview button, that the map becomes available and the states become clickable. This is an example of hierarchical behavior in a user interface.

Every action is associated with an element in the visualization (a button, a map, etc.) that can be interacted with. We call these elements *targets*. Additionally, each action can have other attributes such as the type of action, and possible children nodes that can be interacted with once the action is taken. The following are the set of attributes that Loom keeps in an action tree.

- **ActionType (String)** represents the type of action that can be performed on a target. Examples are "clicking", "brushing", "3D rotation", etc. The ActionType string corresponds to the interactions specified internally or interaction plugins written by users.
- **Shape (SVG format)** represents the position and shape of the region that the interactive element spans on the screen.
- **Children (Object)** represents a list of child elements that can be interacted with once this action is taken. An example of children elements are buttons within a dropdown for the parent dropdown button. The dropdown button must first be interacted with before the buttons within it can be chosen.

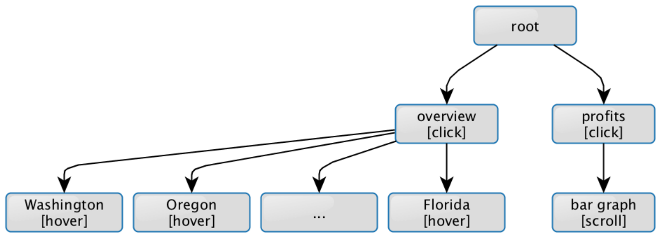


Fig. 3: An action tree for the Tableau superstore example. Every node of the tree is a state in the application. The action that allows one to go to a state is written in brackets inside each node. For example, one can hover over Washington on the map only after they have clicked on the "Overview" button that reveals the map visualization.

Some additional properties such as the name, and description of the target element are also stored to facilitate searching capabilities in the Loom viewer discussed in Section 3.3.

Action trees are specified interactively by users (creators) when creating an interactive Loom recording. Loom provides a toolset to assist users in the tree specification (Section 3.4).

Having modeled the input to the visualization, Loom then systematically collects the static visual responses that are associated with every application state in the action tree. To capture these responses, Loom automatically traverses the tree and takes the actions at each node using a UI bot that controls the mouse and keyboard. After taking every action, Loom takes a screenshot of the visualization and saves this as a visual response. Such automation is generally available by using OS-level libraries.

Every screenshot has a unique ID that is then stored in the corresponding tree node. This ID essentially maps input actions to visual responses in our black-box visualization model.

Starting from the root of the action tree, every branch down to the leaves is a sequence of possible interactions. For instance, in our Tableau example, a user can select the overview, then click on any of the states on the visualized map. Another path is for the user to select the profits button and view the bar graph. Due to this, the traversal algorithm should interact with only the nodes of a sequence starting from the root and ending at leaves and cannot jump to other branches along the way. As another example, consider an interface that requires the user to click on a dropdown, then click on an option of the dropdown to reveal a map visualization. An illustration of its action tree can be seen in Figure 4. The left most branch starts from the root, then based on the left most child of the root, Loom chooses to click on the dropdown. The dropdown opens at this point in time. Loom then has the option to go to the left most child of the dropdown and click on the map visualization button. Finally, Loom can click and interact with the map. Note that after interacting with the map, it is impossible to click on a

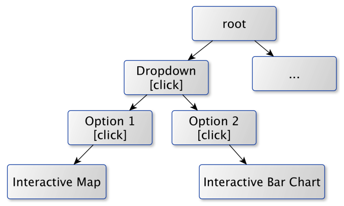


Fig. 4: A simplified action tree for a dropdown button. Different visualizations are shown based on the two options of the dropdown. After interacting with the interactive map, Loom's UI bot needs to start from the root of the tree again in order to reach the interactive bar chart. This requires restarting a pre-order traversal.

different option of the dropdown simply because the dropdown is no longer open. In other words, the application has lost its previous state. Inevitably, Loom must start from the root again then choose a different branch along the way. This process is repeated until all leaves have been visited. This traversal process is equivalent to a pre-order traversal in which after every set of leaf nodes that share a parent, we go back to the root. The algorithm for this traversal is shown in Algorithm 1.

The `DO_ACTION` function in Algorithm 1 performs the specific action for the target (e.g. move mouse, click, etc.). The `CAPTURE_SCREENSHOT` function takes a screenshot and returns an index representing the screenshot. The number is then set to the frame number for the node and will be used at runtime.

Algorithm 1 Traversal of the action tree for automated UI interaction

```
1:  procedure TRAVERSE(tree)
2:      repeat
3:          $n$ ←Find next non-visited leaf node
4:          Find the path from root to $n$
5:          for every $node$ along the path do
6:              VISIT($node$)
7:          for every leaf $node$ that is a sibling of $n$ do
8:              VISIT($node$)
9:      until all nodes have been visited
10:
11: procedure VISIT(node)
12:     DO_ACTION($node.action$)
13:     if node has not been visited before then
14:         $frame\_num$ ←CAPTURE_SCREENSHOT()
15:         $node.frame\_num \leftarrow frame\_num$
```

3.2 Object Construction

A Loom recording consists of an action tree and a visual response per action. In the following, we go over how Loom encodes and stores this information efficiently in a shareable format.

3.2.1 Encoding the Action Tree

Loom stores action trees in JSON format. JSON is chosen because of its wide support in various platforms including web-browsers. The tree's root node is a special node named "window", that does not represent an action but stores the width and height of the recording as well as the rest of the tree as its children. All other nodes contain the action type, the shape of the target, name, description, and the associated ID of the visual response for the action.

Interactive elements can take various shapes in a visualization. We opted for the standard SVG specifications [41] when storing shape information in the tree. For example, rectangular elements have properties such as *x, y, width,* and *height*, while polygons are represented as an array of points with *x* and *y* coordinates. The SVG information is stored in the *shape* attribute of a node.

JSON, and SVG are both heavily used in web development and this has enabled simple browser-support for viewing Loom recordings, increasing shareability, and longevity.

3.2.2 Encoding Visual Responses

Actions are associated with visual responses. Every visual response can be simply stored on disk or in a database. However, to facilitate shareability, it is important to store these action-responses in a packageable, and efficient format that can be opened in various ecosystems and shared across the web. Considering that different states of an application share many similarities and only minimally differ from one another, we used video compression to store the screenshots. Video compression technologies often significantly take advantage of similarities between video frames.

To store the association between the responses and states in the action tree, we stored the frame numbers of each screenshot in the video in its corresponding element in the action tree as the `frame_no` property. The `frame_no` is essentially the aforementioned ID of the visual response. In other words, at this point, Loom has linearized the various application states and stored them sequentially in a video.

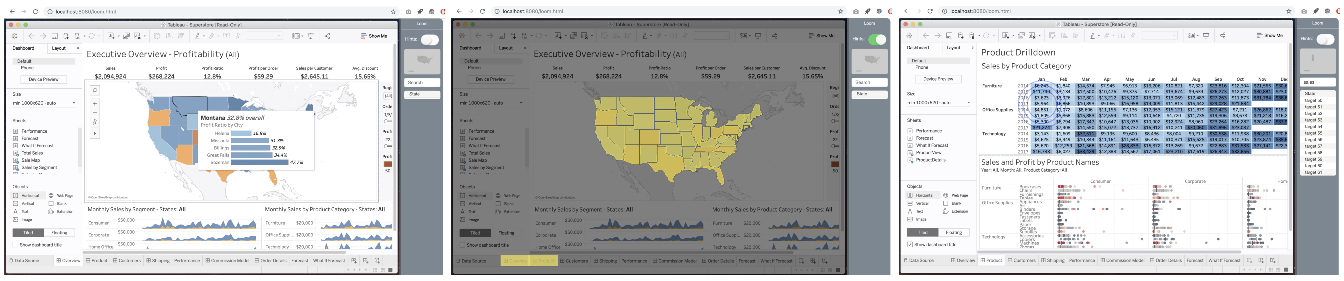
For the video format, we chose MP4 with H.264 compression using the *ffmpeg* implementation. Other technologies such as WebM and H.265 could also be used.

3.3 Reconstructing Interactive Visualizations

Reconstructing the application as an interactive visualization takes place in a browser. The code for viewing a compressed Loom object is written in Javascript and therefore can be executed using any modern browser on different devices such as desktops, tablets and mobile phones.

In order to reconstruct the application's interface, the Loom viewer requires the video that includes visual responses as well as the action tree in JSON format. Then, the viewer essentially provides out-of-order playback of the constructed Loom video, based on user interactions.

Initially, when the viewer is opened, it traverses the action tree and creates an invisible HTML SVG object for each of the targets that the creator has made. The SVG object takes the shape of the target using the shape attribute in the corresponding tree node. These SVG objects will be responsible for handling user interactions. Then, based on the actions of the targets, appropriate event handlers are set up. For example, consider a target with the click action and a respective position P and shape S that describes the selected area in the application. To add this target, Loom adds an invisible SVG element in the DOM with the position and shape of P and S. Loom then adds a Javascript "click" handler. The SVG element is then associated with its node in the action tree using a hash table. When the SVG element is clicked on, it finds the connected node in the tree, takes the frame number associated with the node and seeks the Loom video to the corresponding frame number. Consequently, the image shown to the user is the same as


Executive Overview - Profitability (All)
Product Drilldown
Sales by Product Category
Sales and Profit by Product Names
Monthly Sales by Segment - States: All
Monthly Sales by Product Category - States: All
Montana 32.8% overall

(a) (b) (c)

Fig. 5: (a) The Loom viewer is shown within the browser. The right toolbar provides a mini-map that shows the position and shape of interactive elements. (b) shows the usage of the hints toggle button, highlighting interactive elements. (c) Searching through target descriptions populates a list of possible visual elements in the toolbar. Clicking on a target makes Loom navigate to the appropriate application state and highlights the searched target with a ripple effect (in blue).

what they would have seen if they had interacted with that button in the original visualization.

Although an SVG element is created for every target, not all targets should be interactive at all times. For example, the options of a dropdown should not be available before the dropdown is opened (clicked on). As another example, the SVG elements created for the states in the Tableau example should only be clickable if the user has previously activated the map visualization. In other words, an application state should only become accessible if its parents have been acted on beforehand. Loom handles this using the concept of state machines.

In the reconstruction stage, the action tree is converted to a state machine. The conversion between the action tree model and the state machine is done using the following rules.

1) An application state is created for every node in the tree
2) A transition is created between every state S and its children in the tree. The condition of the transition is set to the action belonging to the child node
3) A two-way transition is created between every two leaf nodes that have the same parent

Within the state machine, every state can be reached if its parent is the current state and its action is executed. Additionally, the root of the tree can always be accessed if the user clicks on anywhere outside of other targets. This is so that users can continue interacting with the system once they have reached a leaf node.

The number of SVG elements that handle interaction can grow very quickly depending on the complexity of the visualization. Additionally, some of the SVG elements can overlap on the screen in condense visualizations. In this case, elements that are on top prevent bottom elements from receiving user events such as clicking. The state machine solves this issue by raising the elements that should be accessible and lowering those that should not be accessible. This is done through CSS by changing the `z-index` style attribute of the elements.

3.3.1 Guided Interaction and Provenance

A user may choose to not capture the entire visualization application and only specify portions of it. This means that images shown to users may have inactive sections. The Loom viewer provides several tools that hints at what is and what is not interactive. Figure 5a shows the Loom viewer in the browser. The right panel is added by Loom. The panel includes a mini-map showing a gray overview of interactive regions. Additionally, a *hints* toggle button is provided. When hints are turned on, interactive regions are highlighted in the visualization (Figure 5b).

Having captured the states and elements of a visualization creates a unique opportunity for Loom at runtime. The panel includes a search tool with which users can search through names and descriptions of UI elements using fuzzy searching [42]. When a user clicks on a search result, Loom switches to the appropriate application state and draws the user's attention to the element they searched for with a ripple effect (Figure 5c). The state machine switches to the parent state of the element so that the user can choose to interact with the element or not.

3.4 Typical Creator Workflow

To create the action tree that describes possible user interactions with a visualization, Loom requires the specification of every interactive element in the application that the creator of the recording wants to include. We call these elements "targets".

Rather than having the creator script the interface manually, such as with a general purpose programming language, Loom shows the creator the underlying application and allows them to define the steps visually and intuitively. This is done through Loom's Overlay Application (LOA).

When the creator starts LOA, they are presented with a semi-transparent overlay like the one shown in Figure 6. LOA allows the creating to visually see the application and define areas for various targets. These targets have an associated shape defining their position and area, and an *action* property that specifies what event they perform.

LOA supports rectangles, circles and complex polygons as target shapes. Created targets can be selected using a selection tool. Every target is accompanied by a settings menu where their properties can be set in. The menu appears in the toolbar whenever a target is selected. If multiple targets are selected, all of their properties can be edited at the same time through the same menu. In other words, the changed property (e.g. the targets' parent) changes to the same value at the same time. This simplifies editing a large number of targets. Loom's toolbar is shown in Figure 7 along with a property menu for a selected target.

3.4.1 Assisted Target Specification

As the creator adds more targets, LOA's display can become cluttered, making subsequent selections difficult. To organize these selections, LOA provides *workspace tabs* (shown in Figure 7).

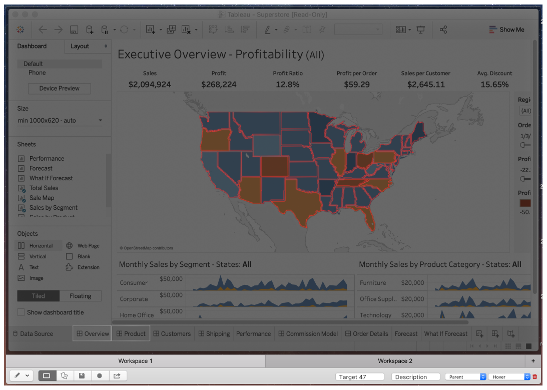


Fig. 6: Loom provides an overlay application (LOA) that allows users to select different parts of their visualization and tell Loom how it should interact with each component. These selections are then used by Loom to construct an action tree that represents the possible interactions with the application. In this figure, the states of the US are target components selected in red.

This way, creators can use each workspace for a specific part of an application. To LOA, all workspaces still define the same visualization. This means that parent target relationships can be specified across workspaces.

LOA also assists creators in specifying large numbers of UI components with a suite of tools, described in the following.

- **Grid Selection.** In many user interfaces, menu items are horizontally or vertically distributed. The grid selection tool simplifies defining a series of rectangular targets based on a single outer rectangle selected by the user and the number of rectangles to automatically create in the X and Y directions.
- **Magic Wand.** Visualization elements can sometimes have complicated shapes making them difficult to select (e.g. a state in the US map). The magic wand tool uses a flood fill algorithm [43] to select a component, based on its boundaries and color. The states in Figure 6 were selected using this tool.
- **Smart Selection.** Visualizations can sometimes have hundreds of visual components. The smart selection tool uses a contour detection algorithm and automatically selects distinct visual elements. The user can then edit or delete the elements as needed. The magic wand and smart selection tools work by taking a screenshot of the underlying visualization, applying their respective image processing method to the image, and adding the selections to LOA.
- **Copy Tool.** UI elements can act differently depending on which target was interacted with before, meaning that the same UI elements can have multiple parents. For example, depending on the value of a radio button, the states on a map might show different information upon hovering. This necessitates adding multiple targets for the same area. The copy tool simplifies this. A user can select a series of targets and by clicking the copy button, new targets of the same shape will be created in a new workspace.

Using LOA on our Tableau example, the creator draws a selection box around the two buttons at the bottom of the screen and sets their actions to “click”. A default name is automatically associated with the selections. The creator also has the option to change those names. In the Tableau application, the creator then navigates to the “Overview” map visualization. Using the Loom overlay, they then select the physical position of each of the states on the map and set the action to “hover”. To convey the hierarchical aspect of the tree, they additionally set the parent node of each of the state selections to the Overview button. This means that the states are only hoverable if the Overview button had been clicked.

To add the bargraph visualization, the creator navigates to the page in Tableau. Using LOA, they then select the portion of the screen that includes the bargraph, and set the action to scroll. By now, the resulting overlay includes all of the selections for our example (Figure 6).

By default, Loom supports several action types such as click, hover, brush, 3D rotation, and sliding. Advanced interactions can be added by writing custom actions in small scripts and are discussed in Section 3.5.

3.4.2 Browser Extension for Specifications

Some visualizations may have many interactive elements and manually selecting them is tedious. For cases when the visualization is web-based, Loom provides a browser extension that assists users in the specification phase. The creator can specify where in the DOM their visualization is and what type of action they are interested in. Then, the extension parses the DOM and returns a configuration file with all of the targets that were found along with their SVG shapes. This file can then be imported in LOA for further editing and enhancement.

The browser extension works by extending the DOM and modifying the “Element” prototype [44] to keep track of the event listeners that the original visualization developer has added. After the page loads, our browser extension then traverses the DOM looking for objects that have event listeners and adds them to the configuration file along with the type of event that the targets support. A JSON version of the configuration is then automatically downloaded to the user’s computer.

3.4.3 Extending Typical Videos

In addition to shareability, encoding visual responses in a video format has also allowed us to enrich traditional videos with interactive components. Resulting videos from this process start with a typical recording showing users using and describing a work. The video then stops and allows the user to interact with portions of the application. Essentially, such videos consist of an initial in-order playback of frames and then an out-of-order playback by Loom’s viewer based on user interactions. This is particularly useful for submitting data analysis and visualization work to conferences, competitions, and as tutorial materials in general.

The video extension option is available as a menu item in LOA, where users can browse a traditional video that loom prepends to the Loom object when in the construction phase. When users append a Loom recording to a traditional video, Loom shifts the `frame_no` attribute of each node in the action tree by the number of frames in the traditional video. The Loom viewer starts by playing the video first until the first `frame_no` in the action tree is reached. It then stops and waits for user interactions.

3.5 Extending Interactions

Visualizations include interactions that are much more complicated than simply clicking and hovering over visual elements. Loom’s interactions can be extended with custom plugins. A Loom plugin

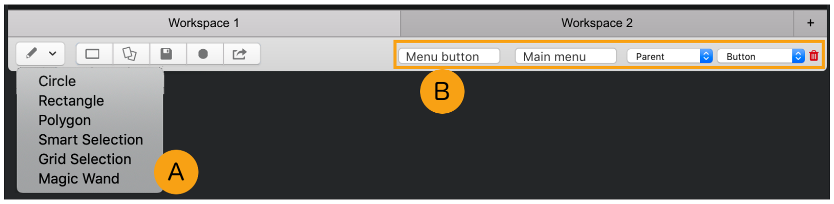


Fig. 7: Loom's toolbar is shown. (a) shows the selection menu supporting circles, rectangles, and polygons, as well as three assistive selection tools. (b) shows the properties of a selected target. The first textbox contains the name of the target. The second textbox contains the description. Two dropdowns receive the parent state and the action of the target from the user. In the current toolbar, two workspaces are opened by the user.

consists of two scripts. The first script tells Loom how to interact with a visual element whose boundary is defined by the user's selection box. In other words, it is essentially a `DO_ACTION` function. The second script tells Loom how to handle interactions in the browser and map them to the appropriate frames in a Loom video. Here, we give an example of how we implemented a plugin for a slider action and a brushing action. Plugins can support even more complex interactions. Section 4.2 explains how we added a plugin for 3D rotation around scientific visualizations.

Consider a slider in a visualization with its knob moved to the left. Given the position and area of a user's selection, a simple slider action moves the mouse to the inner side of the selected rectangle. It then performs a click event, holds the slider knob, and moves it incrementally to the right. At every defined interval, it takes a screenshot. In Loom's viewer, a DOM element is automatically created for event handling as mentioned in Section 3.3. The plugin for the slider sets a Javascript drag event on the DOM target element. Then, based on the position of the mouse in the target, it seeks the Loom video to the correct frame.

Brushing is a common interaction in visualization applications. Consider a rectangular brushable area A. Users can pick a location in A and press the mouse button. They can then drag the mouse elsewhere within A and finally release the mouse. They have essentially selected a box S that can be defined with a start and an end position. To support brushing, the brushing plugin discretizes the interaction. In other words, it divides area A horizontally and vertically into a set of cells. It then starts by capturing every possible combination of selections for the divided cells. For n cells ($\sqrt{n}$ columns and $\sqrt{n}$ rows), a total of n^2 selections can be made. For example, an area that is divided into 16 cells (4 columns and 4 rows) leads to 256 different selections. The captured frames are then linearly indexed and added to the Loom object. In Loom's viewer, a DOM element is created for handling the brushing. The handler registers mouse press and releases, calculating the cells encompassed by the user's brushing. Based on the starting cell and ending cell of the selection it re-calculates the linear index of the suitable frame and seeks the Loom video to the frame.

This technique can be used to support many other interactions such as panning, scrolling, dragging, etc. The general mechanism is to capture possible image responses from the visualization and index them linearly, and finding the linear index based on interactions in the viewer to seek to the correct frame. Loom's current code-base includes support for *clicking, hovering, sliding, brushing, and 3D rotation.*

3.6 Control of Privacy

Typically, the control of privacy is related to the data. Many visualizations that could be public are not shared, simply because their data sources cannot be shared due to various regulations and policies. Sometimes, we see non-interactive visualizations of a protected dataset, but never see an interactive version online, simply because the website would require direct access to the dataset.

Separating a visualization from its data creates an opportunity to look at the privacy and security aspect of visualizations. An interactive Loom visualization only contains images, making it safer to share. Additionally, it is possible to encrypt certain frames and only enable them for authorized users, providing a finer control on privacy.

In our current prototype, Loom provides a command-line program that uses AES encryption to encrypt video frames. The creator of the video can select application states using a query on their target description and choose to encrypt those frames. The frames are extracted from the video into a separate compressed video file, that is then encoded using AES. Within the original Loom video, the frames are substituted with black images. The application states are also tagged in the action tree as unavailable. Hence, the remaining Loom video prevents the extracted interactions, unless patched with the decrypted file.

Loom uses the OpenSSL implementation of AES, however other types of encryption techniques can also be used. Loom's encryption is currently provided as a command line program. A sample command for encrypting the application states that have to do with the "map" keyword in the Tableau example can be `./encrypt.sh loom.mp4 map <password>`. The password must be provided later for decryption.

While a data-connected visualization can also encrypt its dataset, it will be required to decrypt all or most of it at runtime to create the interactive visualization. However in Loom visualizations, a frame will only need to be decrypted if the user has authorization to view it.

4 Case Study

Loom's approach to capturing visualizations is software-agnostic and can support various types of visualizations whether run on desktops or on the web. In this section, we show examples of Loom capturing both information-visualizations, and scientific visualizations. Additionally, we discuss the tools a user would use in LOA in order to capture such visualizations.

4.1 Capturing Information Visualizations

4.1.1 Tableau

Figure 8 shows a more complicated version of the Tableau superstore example in Section 3. The final Loom object included 72 frames with an interactive US map visualization, 10 tabs with various static visualizations, and a dropdown with 3 buttons. The resulting object had a size of 2.6MB at a resolution of 2560x1600. In another example, we also captured a scatterplot with support for brushing. The brushable area was divided into 6 cells across and 6 cells down automatically by LOA. The resulting object included 1368 frames with a size of 6.3MB.

To capture the tableau examples, we used the rectangle tool in LOA for the tabs, and the dropdown. The scatterplot area was also a single rectangle with its action set to "brushing". The interactive map included irregular areas, so we used the magic-wand tool and simply clicked on each state once to create a target for it.

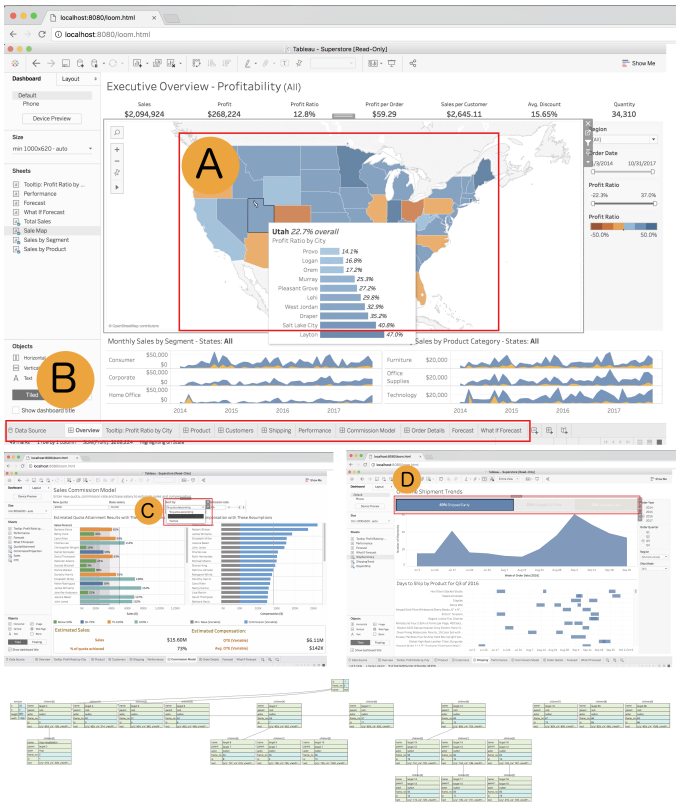


Fig. 8: A Loom object of the Tableau superstore dataset is shown in the browser. An interactive map and clickable tabs are shown in (A). A functional dropdown that changes the order of the data is shown in (C). Three different line graphs can be picked in (D). The action tree of the Loom object can be seen at the bottom. All of the interactions resulted in a 2.6MB file with 72 frames at a resolution of 2560x1600.

4.1.2 Online Visualizations

While online visualizations are already shareable, they are occasionally temporary, especially when they rely on server-side code for retrieving or processing data. For example, visualizations used in online data journalism are often bound to a news agency's servers and can be taken down at any point. More importantly, some technologies become obsolete and the visualization created with them become unusable. One example is the discontinuation of Adobe Flash and its lost support in modern browsers. Throughout the years, many visualizations have been created using Adobe Flash. Loom recordings can provide a way to save, store, and share portions of such applications. This speaks to the "sharing with the possibility of archiving" aspect of Loom since sharing digital information inherently surrenders the control on when the information will be used by others. Figure 9 shows LOA on top of a Flash-based information visualization from the Senseable City Lab at MIT [45].

The visualization shows medical record data. The circular keywords in the visualization have been captured with the assistance of the smart selector. The complete Loom object includes 336 targets created in 3 workspaces. The size of the Loom object is 4.2MB.

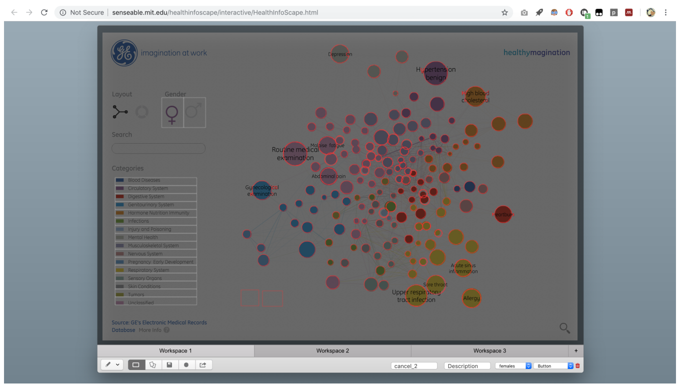


Fig. 9: The Loom Overlay Application is shown on top of an Adobe Flash visualization. The Loom object includes 336 selected targets and has a size of 4.2MB.

While the smart selector simplifies capturing a large number of elements, it can still be time-consuming. However, for web visualizations that utilize DOM elements we can use Loom's browser extension as described in Section 3.4.2. Figure 10-right shows a visualization from the New York Times [46], which has been captured using Loom and re-opened in a Chrome browser. The complete graph in the visualization has 628 interactive nodes that highlight connected neighbors when hovered on. The size of the resulting Loom object is 5.5MB. Figure 10-left exposes the boundary of the selections from the browser extension.

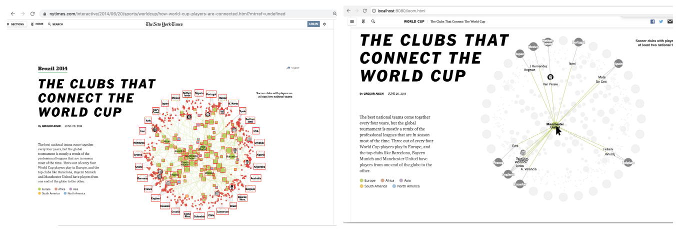

THE CLUBS THAT CONNECT THE WORLD CUP
THE CLUBS THAT CONNECT THE WORLD CUP

Fig. 10: UI targets have been automatically captured and highlighted in red using Loom's browser extension for the 2014 Soccer World Cup visualization from the New York Times (left). The Loom object has recreated the interactive experience in the Loom viewer (right). The visualization no longer depends on the original website or the data source, yet is fully interactive.

4.2 Capturing Large Scientific Visualizations

To showcase Loom's capability on capturing interactions with scientific visualization, we picked the Paraview application as a subject, and opened a volumetric heptane dataset in Paraview. The dataset had a size of 105MB. Figure 11 shows Paraview with the heptane dataset reconstructed within a browser.

One of the most widely used types of interaction in scientific visualization is 3D rotation around an object. To support this, we added a custom action as an extension. In the capture stage, the action simply uses the mouse to exhaustively drag around an object in Paraview in an organized way. The action first rotates the object so that the camera looks at the zenith. It then rotates the object around the X axis towards the nadir. Loom takes screenshots along the way. Going from zenith to nadir once spans 180 degrees. The action script then re-centers the object, and incrementally rotates the object along the Y axis and continues the first step again. This process continues until the complete object has been captured. Figure 11 contains two fully rotational targets in the middle of the screen.

In cases where one has access to the underlying application's API, one can rotate the camera or the object with incremental angles in the custom action script. However, in the case of this example, we aimed for an application-agnostic way of capturing the rotations around an object. Due to this lack of access to the underlying application in this case, we initially measured how many pixels the cursor must travel to complete 180 degrees around the object in our Paraview instance, and then used this to complete the rotations in our custom script. Our action script takes 500 images around an object. That is 20 intervals around the object, each of which includes 25 images from the zenith to the nadir of the object.

In the reconstruction stage, the extension implements a standard arcball algorithm that maps mouse movements to the 500 captured images based on the yaw and pitch of the arcball algorithm. In our example with Paraview and the heptane dataset, we included two sets of rotations, one for a volume rendering and one for a surface rendering. Figure 11 shows the reconstructed Paraview interface within a Chrome browser. What the user sees in the browser is a single frame of the Loom video showing a screenshot of Paraview. Clicking on options B, and C switch between volume rendering and surface rendering. For each option, the rendering in the middle of the screen updates appropriately and can be rotated. As a user drags the mouse cursor on the rendering, their mouse movement is converted to angles using the arcball algorithm. The angles are then mapped to the appropriate image among the 500 captured images from the object, and the image is shown to the user. It is important to note that this is a quantization over the possible rotations around the data and is less smooth than the original experience. However, it stands as an example of complex interaction reconstructed in the browser using Loom, independent of the original data and application. The final recording was approximately 10MB in size.

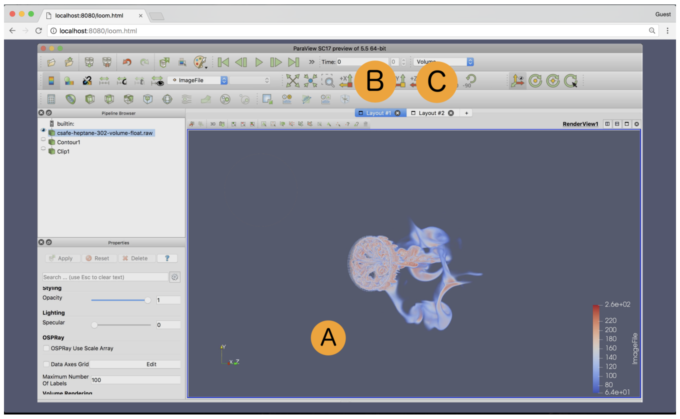


Fig. 11: A reconstructed version of the Paraview interface in the Chrome browser. In this example, the volume can be freely rotated using Loom's arcball extension (A). The two tabs at the top (B, C) change between volume rendering and surface rendering modes. The Loom object for these interactions is approximately 10MB.

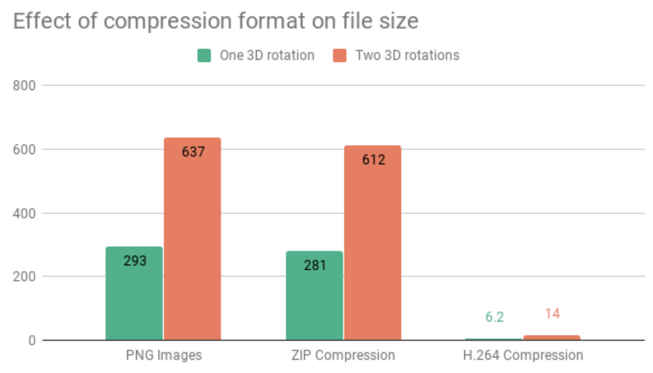


Fig. 12: The effect of H.264 compression on two Loom objects is shown. The size of the object is at a minimum, 38 times smaller than the raw images. This is mainly due to the similarity between consecutive frames.

5 Results and Discussion

In contrast to typical videos, subsequent frames in Loom recordings are extremely similar to one another. This results in great compression of the frames and small sizes. Figure 12 compares the compressed and uncompressed versions for two Loom recordings that included volumetric visualizations in Paraview. We can see that Loom recordings are 38 times smaller when compressed, compared to the raw images.

The size of Loom recordings change not based on the size of the visualized data, but by the amount of interaction that a user needs. This provides an alternative control on the size of visualizations.

Based on the definitions in [21], an interaction is an action by a user with an intent to change the state of an application. Every frame in a Loom video is a new visual response to a state change. Therefore, to quantifiably measure the amount of interaction Loom provides, we consider each frame an interaction.

Although increasing the number of interactions also increases the size of Loom recordings, it can also help with video compression. In Table 1, the Loom video size and the number of interactions for five different test cases is shown. Additionally, the KB/Interaction ratio shows how much an interaction is taking space in the Loom recordings. The Tableau examples have much less interaction and subsequently less number of frames. However, their KB/Interaction ratio is much larger than the Paraview examples that compress better. This is simply because the frames that involve 3D rotation are not drastically different from one another.

In the Table, we can also see the effect of adding new types of interactions. The last row shows an example that includes a clipping interaction that clips a surface rendering by sliding the mouse. The added detail of the surface renders has affected the KB/Interaction ratio.

Frame encryption also affects file size. While Loom compresses extracted frames first and then encodes them with AES, the size of the Loom video plus the size of the encoded file are larger than the original unencrypted Loom video. In our tests, the resulting two files were approximately 1.1 times larger than the original Loom video.

TABLE 1: Results of comparing a Loom object's video size and the number of interactions it provides (measured as new frames). The similarity between frames in cases where there is more interaction has contributed to how well the video compresses. In all cases, the cost of adding an interaction to the object was less than 40KB.

Test Case	**Loom Video Size (MB)**	**Number of Interactions**	**KB/Interaction**
1) New York Times	5.5	628	8.96
2) Tableau Superstore	2.6	72	36.97
3) Tableau Superstore (with brushing)	6.3	1368	4.6
4) Senseable City Lab (Adobe Flash)	4.2	336	12.5
5) Paraview (two 3D rotations)	9.9	1003	10.10
6) Paraview (two 3D rotations + clipping)	14	1015	14.12

TABLE 2: Comparison of recording effort and the number of specified target elements. Note that the difference between the Tableau Superstore example with and without brushing is 1296 frames but only one extra target for the user. Examples with more targets however, take more user specification time such as the Adobe Flash example.

Test Case	**# Targets**	**Avg. Time (m)**
Senseable City Lab (Adobe Flash)	336	18.16
Tableau Superstore (with brushing)	73	10.2
Paraview (two 3D rotations)	6	3.41

Despite these relationships, it is important to note that in all of our tests, the ratio was always below 40KB per interaction.

In addition to size, it is important to look at how long it takes for Loom to record visualizations. Loom's UI-bot periodically waits after every interaction in order to let the underlying visualization update if it needs to. Therefore, generating Loom recordings takes time depending on the number of interactions. The most time consuming case in our examples was for the Paraview object with two full 3D rotations and a surface clipping slider action. Loom's UI-bot took approximately 40 minutes to capture the interactions. That is less than 2.5 seconds wait time between every two interactions. The duration can be changed in Loom.

5.1 Recording Effort and Feedback

As with creating videos, creating Loom recordings can also take a considerable amount of time and effort by the creator. Interested in how long UI-specification takes for various visualizations, we asked 3 computer science students to use LOA and specify the states for 3 examples in this paper (test cases 2, 4, and 5 from Table 1). We then timed their work. Every student worked on two examples.

Before the experiment, we took 15 minutes to explain 1) how Loom works, and 2) the example they would capture. On average, the Adobe Flash example took 18.16 minutes and the students were able to specify approximately 376 states. They initially used the smart selector to automatically select the different circular targets and used the circle tool to specify some that had not been captured automatically. The Tableau example (with 72 states) took 10.2 minutes on average and the Paraview example, took the least amount of time for specification (only 3.41 minutes) since it only required a few rectangular selections.

When asked about their experience in using LOA, one of the students mentioned that their experience with imaging software such as Adobe Photoshop helped them recognize and use LOA's toolset. Additionally, another student noted that LOA can significantly benefit from keyboard shortcuts such as hitting the "Enter" key for approving a selection instead of clicking on the approve button which can be tedious.

The amount of time it takes for users is not necessarily correlated with the number of interactions. Complicated interactions such as brushing and 3D rotations for example, only take a few mouse strokes. Recording effort mostly correlates with the number of UI targets in the application as that is what the users are truly selecting (Table 2). While smart tools in LOA alleviate this task, it can still be time consuming to correct or adjust the automatic specifications for a large number of targets.

5.2 Discussion

5.2.1 Limitations

Capturing an interactive experience without the original code and data can induce some limitations on the types of interactions possible. Similar to videos, while many application states can be captured, there can only be a finite amount, making it impossible to completely replace Turing complete code. Therefore, it is important to discuss what is and what is not possible to capture with Loom.

Many works have introduced interaction taxonomies and organized the types of interactions used in visualizations. With regard to the taxonomical *dimensions* of interaction [48], Loom supports *stepped*, *passive*, and *composite* interactions and does not support *continuous*. As mentioned in Section 3.5, some continuous interactions can be imitated with discrete alternatives. For example, scrolling can be discretized such that the application scrolls in steps. We classify Loom's interaction *types* with regard to the taxonomy of Brehmer et al. [47]. Table 3 shows the results. In essence, Loom supports interactions that are pre-determined and discrete.

While this taxonomy defines the theoretical limits of Loom, in practice, exploratory visualizations such as zooming and navigating maps can exponentially grow the number of frames. For example, the discretized brushing in our Tableau example quickly grew the number of frames to 1368 from 72. However, it is important to note that Loom's goal is not to fully replicate a visualization, but to provide a solution for sharing interactive recordings without their data, and is more suitable for explanatory visualizations.

5.2.2 Suitability for Visualization vs. Other Applications

Visualization applications often rely on external and pre-defined data sources. This creates a unique opportunity for systems like Loom. Loom cannot be used with general utility applications such as Microsoft Word simply because their data source is provided by users (i.e. text) at runtime and is not pre-determined. Moreover, the types of interactions with visualizations are well- and pre-defined, making capturing much simpler. General applications on the other hand support interaction with components that are created on the fly based on user data.

TABLE 3: Loom's support for different types of interactions based on the taxonomy of Brehmer et al. [47] is shown. In general, discrete interactions can be captured, while continuous and undetermined interactions cannot be captured by Loom.

Taxonomical Unit	Support	Comments
Select	✓	Discrete Selections Supported
Navigate	✓	Navigations such as panning are supported if discrete
Arrange	✓	-
Change	✓	-
Filter	✗	Filtering is usually undetermined (based on user input)
Aggregate	✗	Aggregation typically exponentially increases the state space

6 Conclusion

In this paper, we presented Loom as a system that allows users to create interactive recordings of visualizations that can be shared without the original data or software. Loom essentially bridges the gap between dataless but non-interactive sharing of visualizations (such as images and videos), and data-dependent but interactive sharing (such as sharing the data source and application itself).

We showed examples of using Loom to capture several types of interactions (e.g. clicking, hovering, sliding, brushing, and 3D rotation) together with feature-rich scientific visualization, and function-rich information visualization. We then discussed the types of interactions that Loom supports with regard to well-known interaction taxonomies and reported on the timings and feedback of users.

Resulting Loom objects are fully interactive but have zero dependency on the original data and software that created the visualizations. On average, at retina display resolution (2560 x 1600), the Loom objects have an average storage cost far below 50KB/Interaction. The Loom viewer is Javascript based, compact, and runs within any modern web browser.

Our current work has a few limitations. First, even though we have tested on the types of visualization and confirmed efficacy, it's not yet conclusive to what level Loom will work for all visualization components that exist. Some types of interaction can be harder to automate. Integrating more automatic and AI-assisted approaches could be a path forward. Second, Loom is good for sharing explanatory visualizations, as opposed to exploratory. The latter often explores a large exponential space that quickly grows in size when rasterized. We plan on tackling these directions in future works.

Loom's use of standard H.264 allows recordings to be appended to traditional videos. While the current version supports appending one traditional video to another Loom recording, more complex combinations of interaction + video could be further studied. Moreover, the impact of interaction + video on a viewer's experience of visualization sets another direction for future works.

Additionally, it is worth noting that Loom is a light-weight solution that can be used for asynchronous distributed collaboration, reproducing user experiences, and/or archiving. Loom itself does not address provenance, although it could be used together with other provenance systems. Further exploring this possibility should be a fruitful direction of future work.

Acknowledgments

The authors would like to thank the anonymous reviewers of this and previous versions of the manuscript for their valuable comments and suggestions. The authors are supported in part by NSF Award CNS-1629890, and USDI-NPS P14AC01485.

References

[1] B. Van der Haak, M. Parks, and M. Castells, "The future of journalism: Networked journalism," *International journal of communication*, vol. 6, p. 16, 2012.

[2] S. K. Badam and N. Elmqvist, "Polychrome: A cross-device framework for collaborative web visualization," in *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*. ACM, 2014, pp. 109–118.

[3] C. Donalek, S. G. Djorgovski, A. Cioc, A. Wang, J. Zhang, E. Lawler, S. Yeh, A. Mahabal, M. Graham, A. Drake *et al.*, "Immersive and collaborative data visualization using virtual reality platforms," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 609–614.

[4] Y. Demchenko, P. Grosso, C. De Laat, and P. Membrey, "Addressing big data issues in scientific data infrastructure," in *Collaboration Technologies and Systems (CTS), 2013 International Conference on*. IEEE, 2013, pp. 48–55.

[5] J. Thomson, D. Adams, and K. Walker, "Metadata's role in a scientific archive," *Computer*, vol. 36, no. 12, pp. 27–34, 2003.

[6] J. Steele and N. Iliinsky, *Designing Data Visualizations*. O'Reilly Media, Inc., 2011.

[7] Z. A. King, A. Dräger, A. Ebrahim, N. Sonnenschein, N. E. Lewis, and B. O. Palsson, "Escher: a web application for building, sharing, and embedding data-rich visualizations of biological pathways," *PLoS computational biology*, vol. 11, no. 8, p. e1004321, 2015.

[8] A. Graves, "Creation of visualizations based on linked data," in *Proceedings of the 3rd International Conference on Web Intelligence, Mining and Semantics*. ACM, 2013, p. 41.

[9] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.

[10] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 341–350, 2016.

[11] P. Hunt, P. O'Shannessy, D. Smith, and T. Coatta, "React: Facebook's functional turn on writing javascript," *Communications of the ACM*, vol. 59, no. 12, pp. 56–62, 2016.

[12] N. Jain, A. Bhansali, and D. Mehta, "Angularjs: A modern mvc framework in javascript," *Journal of Global Research in Computer Science*, vol. 5, no. 12, pp. 17–23, 2015.

[13] National Park Service, "Species-Mapper Visualization," 2019 (accessed September 12, 2019), https://maps.nps.gov/species/.

[14] Adobe, "Flash & The Future of Interactive Content," 2019 (accessed Jan 16, 2019), https://theblog.adobe.com/adobe-flash-update/.

[15] J. Heer and B. Shneiderman, "Interactive dynamics for visual analysis," *Queue*, vol. 10, no. 2, p. 30, 2012.

[16] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Enabling interactive multiple-view visualizations," in *Visualization, 2005. VIS 05. IEEE*. IEEE, 2005, pp. 135–142.

[17] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *The visualization handbook*, vol. 717, 2005.

[18] E. T. Stanton and W. P. Kegelmeyer, "Creating and managing" lookmarks" in paraview," in *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. IEEE, 2004, pp. p19–p19.

[19] L. Moreau, "The foundations for provenance on the web," *Foundations and Trends in Web Science*, vol. 2, no. 2–3, pp. 99–241, 2010.

[20] W3C, "An Overview of the PROV Family of Documents," 2019 (accessed January 23, 2019). [Online]. Available: https://www.w3.org/TR/prov-overview/

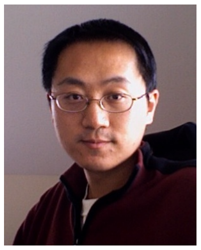
[21] I. Suriarachchi, Q. Zhou, and B. Plale, "Komadu: A capture and visualization system for scientific data provenance," *Journal of Open Research Software*, vol. 3, no. 1, 2015.

[22] Y. L. Simmhan, B. Plale, and D. Gannon, "A framework for collecting provenance in data-centric scientific workflows," in *Web Services, 2006. ICWS'06. International Conference on*. IEEE, 2006, pp. 427–436.
[23] A. Camisetty, C. Chandurkar, M. Sun, and D. Koop, "Enhancing web-based analytics applications through provenance," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 131–141, 2019.
[24] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala, "Graphical histories for visualization: Supporting analysis, communication, and evaluation," *IEEE transactions on visualization and computer graphics*, vol. 14, no. 6, 2008.
[25] T. Grossman, J. Matejka, and G. Fitzmaurice, "Chronicle: capture, exploration, and playback of document workflow histories," in *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM, 2010, pp. 143–152.
[26] H.-T. Chen, L.-Y. Wei, and C.-F. Chang, "Nonlinear revision control for images," in *ACM Transactions on Graphics (TOG)*, vol. 30, no. 4. ACM, 2011, p. 105.
[27] K. Stockinger, J. Shalf, K. Wu, and E. W. Bethel, "Query-driven visualization of large data sets," in *VIS 05. IEEE Visualization, 2005.*, Oct 2005, pp. 167–174.
[28] M. Glatter, J. Huang, S. Ahern, J. Daniel, and A. Lu, "Visualizing temporal patterns in large multivariate data using modified globbing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1467 – 1474, 2008.
[29] J. Zhou and M. Takatsuka, "Automatic transfer function generation using contour tree controlled residue flow model and color harmonics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1481–1488, 2009.
[30] B. P. Nguyen, W.-L. Tay, C.-K. Chui, and S.-H. Ong, "A clustering-based system to automate transfer function design for medical image visualization," *The Visual Computer*, vol. 28, no. 2, pp. 181–191, 2012.
[31] M. Raji, A. Hota, R. Sisneros, P. Messmer, and J. Huang, "Photo-Guided Exploration of Volume Data Features," in *Eurographics Symposium on Parallel Graphics and Visualization*, A. Telea and J. Bennett, Eds. The Eurographics Association, 2017.
[32] M. Raji, A. Hota, and J. Huang, "Scalable web-embedded volume rendering," in *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2017, pp. 45–54.
[33] M. Raji, A. Hota, T. Hobson, and J. Huang, "Scientific visualization as a microservice," *IEEE transactions on visualization and computer graphics*, 2018.
[34] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 424–434.
[35] F. Lamberti and A. Sanna, "A streaming-based solution for remote visualization of 3d graphics on mobile devices," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 2, pp. 247–260, 2007.
[36] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.
[37] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.
[38] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game defcon," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 100–110.
[39] R. G. Dromey, "Formalizing the transition from requirements to design," in *Mathematical frameworks for component software: Models for analysis and synthesis*. World Scientific, 2006, pp. 173–205.
[40] "Software Container Platform - Docker: https://www.docker.com/," 2018 (accessed October 14, 2018). [Online]. Available: https://www.docker.com/
[41] W3C, "SVG Specifications 2.0," 2019 (accessed September 8, 2019). [Online]. Available: https://www.w3.org/TR/SVG2/
[42] K. Risk, "Lightweight fuzzy-search, in JavaScript," 2019 (accessed January 23, 2019, https://github.com/krisk/Fuse.
[43] S. Torbert, *Applied computer science*. Springer, 2016.
[44] M. Foundation, "HTML's Element Object Specification," 2019 (accessed September 13, 2019), https://developer.mozilla.org/en-US/docs/Web/API/Element.
[45] "Senseable City Lab - MIT," 2019 (accessed January 21, 2019). [Online]. Available: http://senseable.mit.edu/
[46] NYTimes, "The New York Times visualization on the 2014 World Cup," 2018 (accessed March 28, 2018), https://www.nytimes.com/interactive/2014/06/20/sports/worldcup/how-world-cup-players-are-connected.html.
[47] M. Brehmer and T. Munzner, "A multi-level typology of abstract visualization tasks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2376–2385, 2013.
[48] R. Spence, *Information Visualization: Design for Interaction, 2nd Ed.* Prentice Hall, 2007.

Mohammad Raji Mohammad Raji received a BS and an MS in Computer Engineering from Razi University, Iran in 2008 and 2012, respectively. He received his PhD in Computer Science from the University of Tennessee, Knoxville, in 2019. His research interests include data intensive visualization system architecture, web-based data visualization systems, and deep learning. He joined Google as a full-time technical staff in 2020.

Jeremiah Duncan Jeremiah Duncan received his BS in Computer Science from the University of Tennessee, Knoxville, where he is currently a PhD student. His research interests include machine learning, data visualization, and mixed reality.

Tanner Hobson Tanner Hobson received his BS in Computer Science from the University of Tennessee, Knoxville, where he is currently a PhD student. His research interests include distributed computing, mixed reality visualization, and web-based systems architectures.



Jian Huang Jian Huang is a professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research focuses on data visualization and analytics. He received his PhD degree in computer science from the Ohio State University in 2001. Dr. Huang's research has been funded by National Science Foundation, Department of Energy, Department of Interior, NASA, UT-Battelle, and Intel.