Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation

Joshua Hibschman
Northwestern University
Evanston, IL USA
jh@u.northwestern.edu

Haoqi Zhang
Northwestern University
Evanston, IL USA
hq@northwestern.edu

ABSTRACT

Professional websites contain rich interactive features that developers can learn from, yet understanding their implementation remains a challenge due to the nature of unfamiliar code. Existing tools provide affordances to analyze source code, but feature-rich websites reveal tens of thousands of lines of code and can easily overwhelm the user. We thus present *Telescope*, a platform for discovering how JavaScript and HTML support a website interaction. Telescope helps users understand unfamiliar website code through a composite view they control by adjusting JavaScript detail, scoping the runtime timeline, and triggering relational links between JS, HTML, and website components. To support these affordances on the open web, Telescope instruments the JavaScript in a website without request intercepts using a novel *sleight-of-hand* technique, then watches for traces emitted from the website. In a case study across seven popular websites, Telescope helped identify less than 150 lines of front-end code out of tens of thousands that accurately describe the desired interaction in six of the sites. In an exploratory user study, we observed users identifying difficult programming concepts by developing strategies to analyze relatively small amounts of unfamiliar website source code with Telescope.

ACM Classification Keywords

H.5.2 User Interfaces: Graphical user interfaces (GUI)

Author Keywords

Reverse Engineering; Inspecting; Tracing; Web; JavaScript

INTRODUCTION

Online platforms for learning to code such as StackOverflow, TutsPlus, and CodeSchool attract millions of learners and significantly expand the pool of advanced beginners, yet critical gaps in knowledge and experience remain between advanced beginners and professionals. Current platforms provide few resources for progressing from learning to write functioning code to writing production-quality software. Professional training programs exist, but cost tens of thousands of dollars and are thus not accessible to most. Mentoring and coaching is effective but not currently scalable.

UIST'16, October 16 - 19, 2016, Tokyo, Japan.

ACM 978-1-4503-4189-9/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2984511.2984570

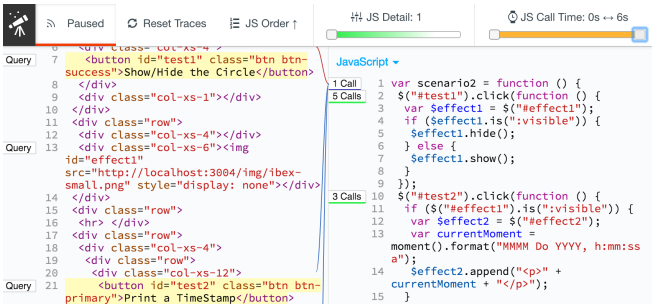

Figure 1. The Telescope platform promotes user discovery of website feature implementations by allowing the user to fine-tune the JavaScript display across time and detail and follow visual links between JavaScript and HTML.

Our work aims to support authentic learning [28] by generating low-barrier learning materials to understand code from professional websites of personal interest. Professional websites offer rich details missing from training examples, content that relates to the real world, and opportunities to think in the models of the discipline. However, despite the abundant availability of front-end code, website source code is difficult to read and can contain superfluous details that distract from learning core concepts.

Deriving learning material from websites presents design and technical challenges due to the magnitude and complexity of the underlying source code. A simple UI interaction may require only ten lines of JavaScript, but modern web production engineering practices make use of libraries and build processes that can push front-end lines of code into the tens of thousands [4, 27, 30]. Bindings between HTML and JavaScript support an interaction, but it is difficult to determine how such bindings are constructed. A simple calendar widget, for example, could be created entirely in JavaScript and appended to the DOM with listeners, or it could be built in HTML and CSS with inline calls to JavaScript hooks. Embedding the widget amidst all its library or utility code in a minification build process blurs the location and scope of code most relevant to enabling the widget's functionality. With existing tools [23, 26, 18, 8, 1, 12, 3], it is difficult to (1) capture the entire scope of JavaScript used, (2) identify the interplay between JavaScript and HTML, and (3) trim away inactive code and library code that get in the way of learning.

We thus introduce *Telescope*, a platform that supports the discovery of website feature implementation by allowing the

user to fine-tune a composite view of responsible JavaScript and explore visual links between JavaScript, HTML, and rendered UI components (see Figure 1). Telescope helps users generate low-barrier learning materials — less than two hundred lines of code — from tens of thousands of lines of complex website code. For example, a curious user could discover how an interactive map component achieves its dragging effect in JavaScript and HTML by setting Telescope's JavaScript detail level to minimum (dom-modifiers only) and time constraints before and after the click-and-drag. By clicking call and query markers in the interface, visual lines connect JavaScript methods to queried DOM elements, and corresponding DOM components are highlighted in the website.

The conceptual contribution of this work is the idea of *helping users understand complex website code by generating low-barrier learning materials*. Telescope introduces three design principles to support this idea:

1. *Single Composite View*: As a user interacts with a website, Telescope brings together relevant JavaScript for an interaction into a single, composite JavaScript view to resolve the challenges in finding all code relevant to a behavior in unfamiliar code [14]. Users can easily hide sources they deem irrelevant or alter the display order of script sources relative to their dependency load order.

2. *Detail and Time Controls*: The user can scope relevant Javascript by call time and control the amount of detail they wish to see, ranging from showing non-library DOM-modifying code only to showing all JavaScript present in the website. These controls address a critical need discovered through our human-centered design process, when we found users struggling to understand the code for an interaction when there is either too little or too much JavaScript to analyze.

3. *Visual Links*: Visual links connect active JavaScript to lines of HTML and website DOM components to expose end-to-end functionality.

The technical contributions of this work support Telescope's design principles and enable using Telescope to examine website UI interactions across the open web in real time. Specifically, we introduce (a) the *Wisat architecture*, which supports source code tracing and instrumentation as well as shared Telescope sessions on public websites, and (b) the *Sleight-of-Hand method* (SoH), which swaps a website's client-side implementation during runtime with its instrumented counterpart. The SoH method transitions websites from a non-traceable state to a fully instrumented state, supporting live interaction traces as a user interacts with their website. The Wisat architecture then transmits runtime traces used to decide which JavaScript is displayed in Telescope's composite view and provides the linking data necessary for drawing connections between JavaScript, HTML, and website components.

In the rest of this paper, we review related work in UI feature discovery and source code inspection. We then introduce Telescope and its main components for tuning UI discovery and linking JavaScript and HTML source code. To examine Telescope's performance and study its effectiveness, we present the results of a case study using Telescope on seven professional websites, and from an exploratory study with five users. We conclude with a discussion of design principles, limitations of our approach, and future work enabled by Telescope.

RELATED WORK

Telescope presents a new method for source discovery that can enable new learning opportunities while addressing a number of the known challenges of learning from unfamiliar code. By helping users to explore professionally written examples of personal interest, Telescope aims to help users overcome the Design, Selection, and Coordination barriers to learning from code identified by Ko et al. [21]. Telescope also extends our ability to practice web foraging [6]. Advancing previous work by Brandt et al. [6], which enabled developers to forage tutorials and reference examples, Telescope opens up opportunities for eliciting examples from across the open Web. Guided by Gross and Kelleher's study [14] on identifying functionality in unfamiliar code, Telescope provides affordances that implement suggested directives such as, "connect code to observable output" and "provide interactions to fully navigate code," to overcome the challenge of finding relevant code.

Existing tools contribute design techniques to highlight, filter, and curate aspects of code responsible for an effect. Theseus [23], FireCrystal [26], Unravel [18], Scry [8], and Clematis [1] each contribute distinct methods of helping users understand the difficult nature of JavaScript execution in web development by highlighting and collecting responsible code, tracking and diffing UI changes, and logging complex operations. Techniques such as Tutorons [17], Gidget [22], WebCrystal [10], Whyline [20], and Dinah [15] curate programming techniques for the user as they explore code provided by the interface. Telescope contributes techniques designed to solve the remaining problem of condensing and linking aspects of code into low-barrier learning material from websites based on user interest.

Visual learning techniques from other works help users to easily see the dynamic effects of their code. Glimpse [11], PyTutor [16], and Bret Victor's "Learnable Programming [31]" demonstrate techniques to visualize how code produces a UI and how code modifies program state as it runs. Most modern web browsers provide affordances for users to find responsible source code through visual breakpoints in the DOM [3], JavaScript beautifiers [12], and HTML change highlights [13]. Telescope addresses remaining difficulties in conceptualizing the relationships between JavaScript, HTML, and DOM components through its visual linking technique.

Telescope's architecture extends works in source code augmentation to provide JavaScript instrumentation and runtime trace analysis across the open web. Telescope deploys Fondue — a Javascript instrumentation strategy introduced by Lieber et al. [23] — as an API in its Wisat architecture. Telescope's *JS Detail* control leverages filtering techniques

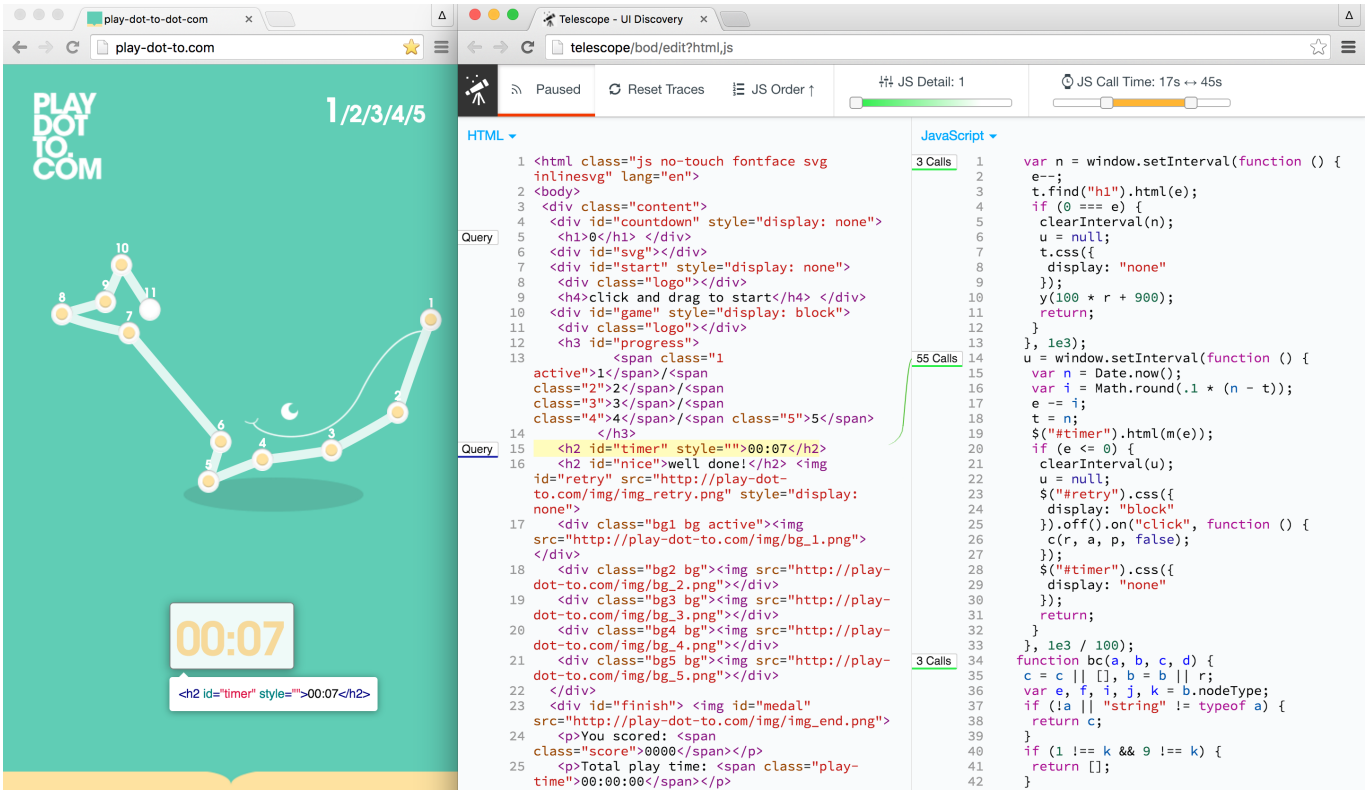

Figure 2. The Telescope interface is being used to discover how this HTML5 connect-the-dot game's timer works. The interface is paused to freeze the current view. The detail level is set at minimum, and the JavaScript call time is constrained between the 17th and 45th second of execution. The left Telescope panel (middle) shows a filtered HTML view, where an active element is highlighted and query markers denote that JavaScript queried those lines during the chosen time window. The right Telescope panel shows the website's JavaScript, filtered by time and detail. With the current settings, only the most relevant JavaScript is displayed: active non-library JavaScript which queried the DOM in the constrained time frame. A curved line is drawn to connect the JavaScript line to its DOM query.

from Unravel's API Harness [18] and Scry's mutation observation [8]. Similar to Maras et al's JavaScript source identification technique [24], Telescope analyzes a source call graph to determine JavaScript involved in modifying the DOM and functions present in supported JavaScript libraries. But unlike Maras et al's browser extension, Unravel, and Scry — which offer a limited view of JavaScript execution such as DOM API callers — Telescope fully instruments a website's source code. Building upon a common goal to work on multiple JavaScript runtimes as in Unravel [18] and Theseus [23], Telescope brings shared feature exploration agnostic of runtime. One website inspection session is shareable to many users over the web, providing the ability for users to explore and learn together in a distributed setting.

Most relevant to our work, Srcy [8] and Unravel [18] allow users to record UI interactions and explore relevant JavaScript underlying the interaction on public websites. We address with Telescope three main limitations of these previous works for understanding web UI implementations. First, Unravel and Scry adopt sequential workflows that require back-and-forth navigation from the interface to individual JavaScript files to obtain what Telescope provides in a single view. As a point of contrast, in Unravel's study, such affordances helped users find the first relevant source more quickly, but did not help them to more deeply understand an implementation beyond that, as Telescope demonstrates in its study. Second, Unravel and Scry examine JavaScript in a scope limited to DOM queries and callers, whereas Telescope's default detail shows DOM-queries and callers, then expands to the global scope, showing more activity such as AJAX and MVC event logic. For example, Telescope could help a user discover how search results are buffered in memory before writing the results to the DOM, whereas the other systems cannot. Finally, Unravel and Scry provide single-direction inferences between a DOM element and the JavaScript that operated on it. In contrast, Telescope provides visual bidirectional links between JavaScript lines and DOM elements, helping users see how multiple DOM elements are affected by a single JavaScript call and vice versa.

TELESCOPE

Telescope is a web-based platform for producing learning material to implement a UI interaction. By using the Wisat architecture discussed later, Telescope receives JavaScript runtime traces and DOM state changes from a website's UI during use. The user views all JavaScript for a website in a single composite view, condensed by time constraints and detail filters. User-activated visual links connect JavaScript, HTML, and rendered components in the browser. (See Figure 2).

Receiving JavaScript, HTML, and Trace Activity

A user launches Telescope by initiating website instrumentation from a browser extension. Once connected, Telescope begins receiving traces and its interface updates in real time to reflect the latest DOM state and an accumulation of JavaScript traces. Queried DOM elements are marked with a *Query* gutter marker. Active JavaScript functions are marked with a *Call Count* gutter marker, a technique we adopted from Theseus [23]. Telescope continuously analyzes call graphs to determine which JavaScript calls were involved in querying the DOM. If an active function is identified as being involved in a DOM query, it is marked with a green call marker instead of a colorless marker to highlight its significance. Depending on the detail setting, a user may see call counts increasing in a specific subset of JavaScript, or across the entire source of the website.

Tuning Telescope: Order, Detail, and Time

A core design goal in Telescope is to avoid overwhelming the user with large amounts of trace information by presenting the most relevant JavaScript together in a single composite view. Most of the websites tested in our case study, such as The New York Times "Snow Fall" article, have tens of thousands of lines of unminified JavaScript and hundreds of lines of HTML. Even a simple photo-slideshow change effect could utilize thousands of function invocations if embedded in MVC logic from a large JavaScript framework like Angular or React.

The controls in the header of the Telescope interface allow the user to fine-tune the source code activity during a UI interaction. From left to right, the user has the ability to (1) pause/resume activity, and to reset the interface to a cleared activity state; (2) flip the JavaScript presentation order; (3) adjust the detail of JavaScript sources displayed; and (4) constrain the time of active JavaScript sources. We discuss each of these affordances below.

1. Pause/Resume and Reset Traces

The Telescope UI updates continuously as the website's UI state changes to show live updates to source code execution. Users can see active JavaScript populate in view, as well as increasing call/query counts next to JavaScript/HTML lines. To freeze the capture state and ignore ongoing functionality, a Telescope user can pause the interface at its current DOM state and JavaScript trace collection. Users can browse and interact with other UI controls during this frozen state, but no new data will be displayed. Upon resuming, Telescope updates to the latest state of the website. Resetting Telescope empties its collection of JavaScript traces and synchronizes its HTML view with the latest DOM state.

2. JavaScript Order

Early pilot studies revealed that relevant source is often found in scripts at the end of a website's load order. The interpreted nature of JavaScript combined with the disorganized nature of website script-loading leads web developers to load scripts with more dependencies last and fewer dependencies first [2]. As a consequence of this dependency pattern, our earlier prototypes often placed the most important high-level JavaScript hidden at the bottom, leaving relevant code out of view. Based on this observation, Telescope by default inverts the load order to display last-loaded scripts first as a heuristic. The composite JavaScript panel in Telescope displays scripts sorted as a whole, so the inner contents of scripts will remain in their original form. The JavaScript order control allows a user to invert the presentation order, e.g. to support cases where our heuristic may not apply.

3. JavaScript Detail

Early pilot studies also revealed that simply showing users all active JavaScript code provided little value. To support discovery, our test users requested variable control over the detail visible. With Telescope's JS Detail slider, a user can control the amount of JavaScript visible. By default, Telescope slides detail to the left extreme (L1), which shows how higher-level JS achieves an effect using library APIs without showing library internals. Low-level DOM API calls are often wrapped by libraries and would be hidden at this level, e.g. a jQuery call $(div) is displayed instead of the DOM API call. Sliding detail to the other extreme will reveal all of the JavaScript for a website. The detail levels include:

L1 (default): DOM API callers and parent callers, not internal library code.

L2: Active JavaScript, not library internals.

L3: Active JavaScript.

L4: All JavaScript excluding known libraries.

L5: All JavaScript.

4. JavaScript Call Time

Telescope users can use timeline constraints to set a start time and end time to see which functions were executed during the specified interval. While JavaScript can execute asynchronously at arbitrary times, users can still slide the time constraints as a way to omit code outside a time interval, such as initial setup code or continuous interval functions.

Linking HTML, JS, and the Rendered DOM

Telescope provides bidirectional visual links between the HTML, JavaScript, and website DOM to provide end-to-end connections from source code to its UI output. Inspired by *Gliimpse* — which creates in-place visual transitions from code to UI and vice versa [11] — these links help form conceptual models of how JavaScript and HTML work together. But unlike Gliimpse, Telescope shows both the source code state and rendered UI simultaneously. Users can visualize how high level functions change many elements (see Case Study: Mac Pro) or how a single element event can trigger many function handlers (see Case Study: Dot-to-Dot). During Telescope sessions, *Query* markers appear in the HTML pane, and *Call* markers appear in the JavaScript pane. Clicking an HTML query marker draws lines to JavaScript functions which query the HTML line. Clicking a green call marker (signifies DOM-query) draws lines to HTML nodes which were queried by the JavaScript line (see Figure 3).

Exploring HTML-JS links in either direction invokes a response in the website, where the rendered DOM nodes are highlighted in the foreground (See Figure 4). If multiple

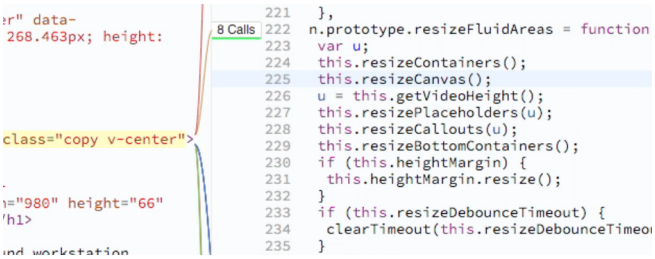

Figure 3. Clicking a Telescope HTML query marker from the Mac Pro website (left) shows lines to four JavaScript functions. In this view, a line leads to function resizeFluidAreas, which resizes elements on scroll.

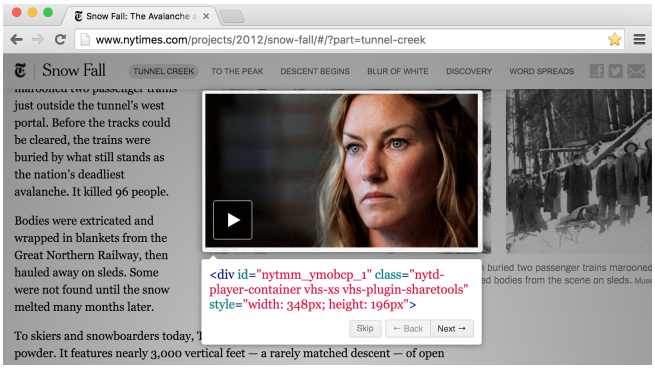

Figure 4. Clicking Telescope's code markers for the New York Times "Snow Fall" website highlights related DOM elements in the website. The DOM element's source is included in the highlight, connecting context to Telescope's HTML view.

DOM nodes are involved in a query, a walkthrough is constructed in the rendered website that highlights each involved element in sequence. Conversely, if elements were deleted, the user is notified that the element is no longer in the DOM.

Hiding Libraries and Irrelevant Scripts

In early pilot studies, traces from library, tracking, and ad-content scripts caused confusion in understanding UI feature implementation. Telescope now hides many libraries and irrelevant scripts and provides affordances for users to hide other scripts they deem irrelevant. By default, library scripts such as jQuery and Angular are hidden, as are popular advertisement and usage-tracking scripts such as DoubleClick and ScoreCardResearch. Users can then hide or show scripts selectively in two ways. Through the JavaScript dropdown view at the top of the JavaScript panel, users can see a complete list of the sources in view and selectively toggle their visibility. Alternatively, users can slide the detail slider to the far right to bring all sources into view (more library and ads) or far left to show only sources relevant to DOM manipulation (less library and ads, see Figure 2 top right).

Design Process and Design Insights

In the process of designing Telescope, we iterated through three software prototypes. Prototype 1 provided the ability to record an interaction and extract a subset of HTML, CSS, and JS into a sharable web sandbox with visual output. Prototype 2 dropped sandboxed output and added affordances to selectively hide inactive code and sources. Clicking JS gutter markers exposed a function's callstack. Prototype 3 gained the Wisat architecture for continuous distributed tracing. After prototype 3, we trimmed features users didn't value and added controls for order, time, detail, and interactive links.

With each prototype we conducted a small pilot study to better understand how to help users overcome learning barriers tied to unfamiliar code. Each study recruited a convenience sample of three junior developers who used the prototype for 30 minutes each and were paid $20. In this process we discovered four primary design insights:

- **Users need variable amounts of JavaScript to understand different programming concepts.** Each prototype provided affordances to selectively trim down the JavaScript, but users were unsure what to trim and found it difficult to remember what they had trimmed from view. Users expressed desires to see both high-level code and low-level utility code at different times to establish a basic understanding of how the program works before looking into its details. We implemented the JS Detail control to adjust the composite JavaScript view to different detail levels such as more minimal for DOM-modifying code or more verbose for deeper discovery involving AJAX and MVC logic.

- **Users have varied processes for playing and inspecting.** Observed in all three studies, some users like to repeat their interaction several times before using Telescope, whereas others will create an interaction and jump to Telescope before it completes. Prototypes 1 and 2 had a static extraction technique that frustrated users who liked to alternate between playing and inspecting. Telescope now continuously updates both its HTML and composite JavaScript as user plays with a website, while also giving the ability to pause and constrain their historical runtime timeline.

- **Users benefit from visual links connecting code to observable output.** Similar to Gross et al's recommendations to *connect code to observable output*, we found that linking the JS and HTML contexts to observable output helped users understand JavaScript's relationship with HTML [14]. By the third prototype, our users were still having trouble understanding how the JavaScript and HTML related even though active-code highlights were provided in both panes. We added support to draw visual lines from either direction between HTML and JavaScript. Upon drawing these lines, the DOM element is highlighted in the website to complete the connection between the code and its output.

- **Metadata and redundant filters overwhelm the user.** Throughout prototype iteration, we kept accumulating features which began distracting users from efficiently using Telescope. To promote simplicity, we cut away features that were distracting or provided little use to achieving the goal of promoting understanding. Features cut included call-stack inspection, the CSS pane, the HTML output pane, code-hiding toggles, and other extraneous features.

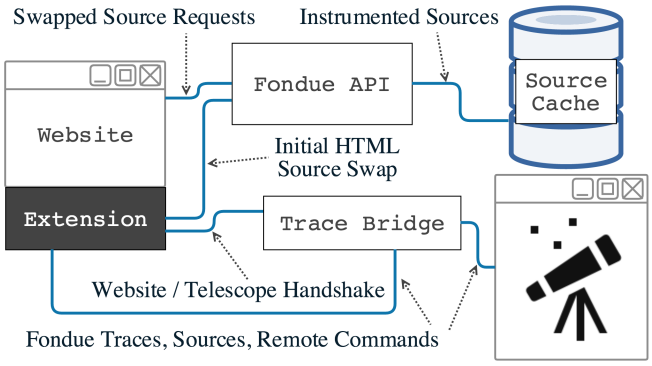

Figure 5. The Wisat architecture supports Telescope's ability to remotely process website interaction traces. A website receives its initial source swap via the Chrome extension. The website fetches instrumented scripts from the Fondue API (top), and the Chrome extension negotiates a two-way handshake via the Trace Bridge to connect it with its Telescope session (bottom). Upon successful connection, JavaScript traces and source data propagate continuously over the trace bridge.

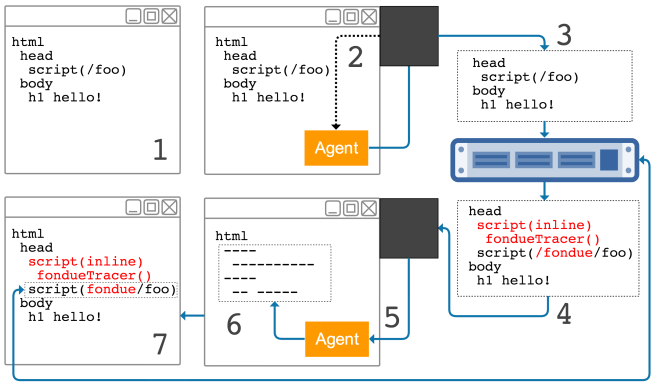

Figure 6. The Sleight-of-Hand technique pictured above is a 7-step process for instrumenting a website's source code via browser extension (black squares) and external instrumentation server (blue, middle right). After website load (1), the extension deploys an agent (2). The agent sends the sources for instrumentation via extension (3), which are returned (4), passed to the agent (5), and swapped for the originals, deleting references (6). The browser makes requests for the newly instrumented sources (7).

IMPLEMENTATION

Telescope's implementation goals include deployability across the open web, full-scope JavaScript instrumentation, and multi-user session support. Unravel [18] and Scry [8] provide JavaScript traces on public websites but limit their inspection scope to DOM-querying JavaScript. Theseus [23] provides full instrumentation but requires a debugging proxy for setup on public websites. To support future empirical field research and promote user adoption, we seek implementations that are easy for users to install with minimal setup. Existing architectures from related systems are designed to only support single-user sessions.

In the rest of this section, we describe the *Wisat architecture* and *Sleight-of-Hand methodology* that together enable Telescope to bring source instrumentation and JavaScript trace analysis to public websites with minimal user setup. Building upon related systems, Telescope brings Fondue's source instrumentation to the open web, augments Theseus' active code markers with interactive links, and uses JSBin's collaborative online editor environment as a foundation [23, 29]. Telescope consists of a component-based architecture where new technologies can be swapped in or integrated later on.

Wisat Architecture

The Wisat (Web interface swap and trace) architecture supports Telescope's JavaScript instrumentation, trace propagation, source transmission, remote control, and sleight-of-hand source swapping. After a website is instrumented via the browser extension, Fondue API, and source cache (see Sleight-of-Hand Method), the browser extension negotiates a two-way handshake between the website and Telescope interface via the *trace bridge*. Once connected, traces, sources, and remote commands can flow freely between the two, populating Telescope's code views and enabling remote DOM component highlighting (see Figure 5). Designed for web scalability, this architecture separates functional components so that each may be distributed across multiple load-balanced instances. The components of this architecture are defined as:

- Telescope UI: A website for receiving source trace activity from an instrumented website, fine-tuning source findings, and sharing with others.
- Fondue API: REST web service for JavaScript & HTML instrumentation and deobfuscation with caching, served over HTTPS to comply with mixed-content policies.
- Trace Bridge: WebSocket server for live cross-origin-compliant transmission of JavaScript traces, DOM changes, and commands between the website and Telescope interface.
- Chrome Extension: Agent injected into website to deploy the Sleight-of-Hand source swap, broker handshake with Telescope interface, and broadcast source activity.

Sleight-of-Hand Method

The Sleight-of-Hand Method (SoH) expands upon techniques from Fondue [23] to bring source instrumentation to public websites. Current methods for JavaScript source tracing either trace only DOM-querying JavaScript [18, 8, 7] or require a man-in-the-middle debugging proxy [23]. Neither of these approaches fits with our goal to fully trace JavaScript execution and make setup simple. The SoH method — deployed from a one-click-install browser extension — implements full JavaScript traceability by swapping the scripts of a website with their instrumented versions. The SoH process is outlined below (see Figure 6):

1. Load a website and initiate SoH.
2. Deploy a JavaScript agent into the website from a browser extension with bidirectional communication.
3. Agent transmits the OuterHTML property of the root DOM element to the instrumentation API via extension, circumventing cross-origin policy.

	Initial Stats			Website On-Load Activity				Website Interaction Lines of Code			
	Class	HTML LOC Total	JS LOC Total	HTML Queried	Active JS	Telescope Default JS	LOC Reduced	HTML Queried	Active JS	Telescope Default JS	LOC Reduced
XKCD 1110	L	77	11,023	31	9,378	4,730	57.09%	1	1,079	49	99.56%
DotToDot	M	34	12,910	3	9,697	5,534	57.13%	23	1,687	115	99.11%
iPhone SE	M	788	53,810	95	8,648	2,341	95.65%	20	907	84	99.84%
Tumblr	H	182	92,070	34	10,504	4,970	94.60%	1	1,839	52	99.94%
Mac Pro	H	794	33,631	248	10,095	1,877	94.42%	68	1,650	934	97.22%
Southwest Air	H	4,140	35,011	143	8,342	4,577	86.93%	1	1,825	53	99.85%
NYT Snow Fall	H	1,458	41,526	2	2,546	521	98.75%	30	522	150	99.64%
Average		**1,068**	**39,997**	**79**	**8,459**	**3,507**	**83.51%**	**21**	**1,358**	**205**	**99.31%**

Figure 7. Results from our case study show the amounts of code Telescope reduces, using time and detail filters to draw distinction between on-load setup code and interaction code. Each website's complexity class is provided (Small, Medium, High). The JS total lines of code (LOC), calculated after normalized unminification, are listed per each website (left) and categorized by all active JS and the default DOM-modifying JS with inner library code removed. In blue (middle, right) the LOC in Telescope's default view for on-load and interaction show the amount of reduction Telescope performs for the user while maintaining relevance. HTML LOC queried are listed, showing the small portion of DOM elements involved in each UI interaction. Interactions include a map-drag (XKCD), a scroll animation (Tumblr), a dot-drag (DotToDot), scroll-driven video sizing (NYT), a load-and-scroll-driven float (iPhone), a scroll-driven product show (Mac Pro), and a date-picker render and select (Southwest).

4. Instrumentation API returns the HTML with inline scripts instrumented and <script> tags with altered "src" attributes pointing to an instrumentation API URL.

5. Browser extension passes the response to the agent.

6. Agent overwrites the existing DOM with an empty root, iterates through non-native window object attributes and deletes them, and calls clearInterval on global interval indices 1 to 999.

7. Agent inserts instrumented pieces of the DOM in a strict order to control script loading to simulate the script load order of the original site.

SoH leverages vulnerabilities enabled by browser extensions, circumventing source alteration protection by overwriting original sources [9]. An SoH is deployed from a browser extension with liberal permissions to modify the page and communicate with third party servers. It does so regardless of logged-in state or HTTPS encryption.

The SoH method works for many websites, but synchronized HTML/JavaScript workflows and Content Security Policies (CSP) can cause problems. For example, if a user kept scrolling for more news to load in a Facebook news feed, the in-memory JavaScript would reflect news list additions and the HTML would reflect the same. If the SoH was initiated after scrolling for more news, the news list HTML would be correct, but the in-memory JavaScript would not have the news list additions. The result would be an odd-UI experience where interactions hit sync-error handling, such as moving the user back to the top of their news feed. Further, any scripts lazy-loaded after the SoH starts and before SoH ends would cause more UI oddities or potentially break the process. Implementing a whitelist CSP successfully blocks the SoH method, because it asks the browser to enforce a strict list of source domains. In all the websites we tested, only Airbnb enforced a CSP. As a fallback, CSPs can be filtered before page-load by debugging proxies [32].

CASE STUDY

To better understand Telescope's capabilities and performance, we used it to identify relevant lines of code, key interaction methods, and implementation patterns across UI features on seven popular websites. This study aims to address the following research question:

RQ1 To what extent can Telescope reduce and scope lines of code for understanding complex feature implementations?

We chose websites with interesting and complex UI features that are not straightforward to understand, that have over ten thousand lines of code. Interactions of interest include a map-drag (XKCD), a scroll animation (Tumblr), a dot-drag (DotToDot), scroll-driven video sizing (NYT), a load-and-scroll-driven float (iPhone), a scroll-driven product show (Mac Pro), and a date-picker render and select (Southwest). We classified websites as light (L), medium (M), or heavy (H) in proportion to their UI complexity and average number of function invocations. For each example, we tracked the minimum usage necessary to discover UI features on the website, while comparing against Unravel as a control.

Fine-Tuning Lines of Code

Telescope supported discovery on the seven websites with minimal tuning regardless of source code size (see Figure 7). We measured the lines of code visible in Telescope during on-load and interaction, normalizing JavaScript and HTML with unminifying preprocessors. Telescope identified each site's large on-load setup processes (521 to 5,534, mean 3,507 LOC), allowing us to easily scope timeline constraints beyond the setup code to yield each interaction's code (49 to 934, mean 205 LOC). Besides the Mac Pro example, **running Telescope on all other websites with the default detail setting yielded 150 lines or less of code that sufficiently explained how the interaction was created in each site.** With 1 to 68 (mean 21) LOC of HTML queried during interactions, the HTML query markers offer a simple starting point for exploration.

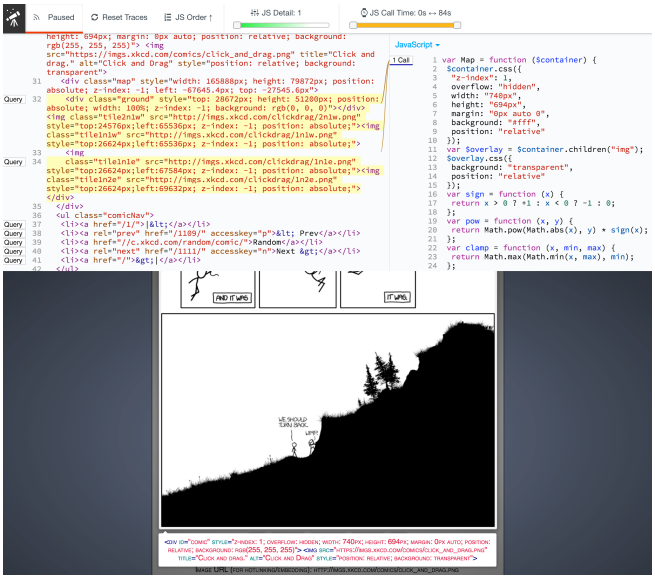

Figure 8. Telescope is being used to discover XKCD's map-drag implementation. A JavaScript call marker has been clicked next to the `Map` function, resulting in HTML line highlights and a DOM element highlight in the website.

Low Complexity Example: XKCD 1110

XKCD's interactive comic #1110 website presents a simple test scenario for Telescope with its relatively small codebase and direct UI interaction (see Figure 8). Telescope revealed a composite 49-line draggable map implementation (excluding library code). We quickly discovered functions `map`, `update`, and `drag` with Telescope's default settings. We examined the startup code and moved the timeline past startup to see the interaction. The map-drag effect is achieved by events bound on `mousedown` that track mouse position relative to a center start position. The map is a grid of image tiles with names representing their position, where images ±1 away from the centered tile are loaded and set to visible, while others are hidden.

Using Unravel on the same interaction, we were able to easily find the same functions behind XKCD's map load, however we needed to look through 420 lines of JavaScript to find how relevant calls in separate files fit together. Unravel showed changes to the DOM caused by dragging the map, but it was difficult to determine the scope of JavaScript operating on the map. Setting DOM breakpoints through Chrome Developer Tools, we were able to step through function calls responsible for modifying the map.

Moderate Complexity Example: Dot-to-Dot

In analyzing the design award-winning Dot-to-Dot game, Telescope helped us to understand how the game connects the dots. We sought to understand the code behind connecting a dot to another: dots appear, a line is drawn, and audio plays a dot sound. We didn't need to look far to find a `dot` class in the setup code, which was referenced later in the JS time 23s to 42s. The JavaScript code was heavily minified, but Telescope expanded it in a way we could infer how functions operated even without their names. Function `c` activates a game round, function `y` starts the timer interval, function `o` draws a line invoking RaphaelJS, and function `v` handles dot clicks and dot animation. Sliding JS detail towards the middle we found a `pop.mp3` xhr request, where the response is stored in a variable and played via `SFX.pop()`.

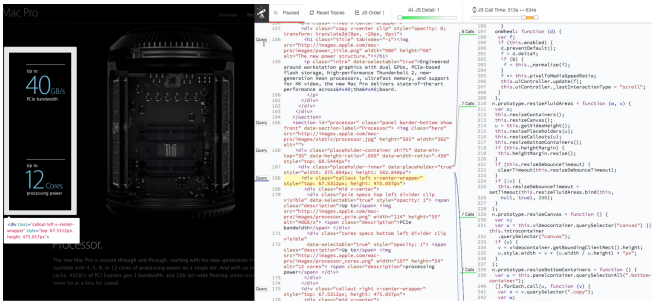

Figure 9. We evaluated Telescope's performance and source discovery on Apple's Mac Pro product demo website. While performance lagged during UI animation, Telescope accurately captured and reduced the source code view to show how the scroll-driven effect works. Above, an HTML line marker has been selected in the Telescope interface that draws lines to linked functions and highlights the DOM component.

In this scenario, Unravel provided hundreds of JavaScript inspection points and DOM changes. We inspected the top two most-called functions and quickly found the game's timer and dot-insertion logic by clicking through Unravel's inspection points. Using Chrome's search feature was more convenient than manually looking through the remaining Unravel results, so we ran find-all queries for RaphaleJS calls and set breakpoints to determine how game rounds began. Separating the game's setup code from runtime code was difficult with Unravel, because all of the JavaScript functions accumulate in one list that is only sortable by call count or function name.

High Website Complexity Example: "Snow Fall"

The Pulitzer Prize winning New York Times article "Snow Fall" stretched Telescope's technical ability with 41,526 lines of JavaScript and 1,458 lines of HTML along with a high volume of recurring background JavaScript execution. In this test, we sought to discover how the Steven's Pass flyover interaction was activated. We scanned through 300 lines of irrelevant ad and tracking code before finding the right Telescope settings. We set the JS Call Time to 41s to 73s and set the JS Detail to the middle, where we found relevant functions `videoBG.setFullscreen`, `checkArticleProgress`, and `percentTillNext` related to an HTML5 video player (see Figure 4). The latter two run on every scroll event and the former is activated when the article progress reaches the "Tunnel Creek" narrative. We found related HTML elements `div.nytmm_video_player`. Unravel quickly revealed results pointing to functions responsible for setting full screen and initiating video playback, but like in the previous case, the magnitude of function traces occluded the search for other meaningful functionality. We were unable to quickly find the remaining functional pieces for checking the article progress and activating new sections.

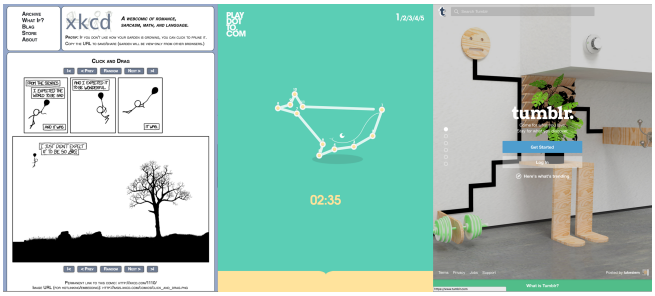

Figure 10. We observed Telescope's use while discovering a map-drag interaction on XKCD (left), a dot-connect interaction on Play-Dot-To.com (middle), and a scroll animation on Tumblr (right).

High Complexity Example #2: Mac Pro

The interactive product page, which disassembles an Apple Mac pro on user scroll, tested Telescope's performance limitations but revealed insight into the website's design (see Figure 9). The initial product-rising animation was captured in Telescope, logging 30k+ function invocations. We scrolled down to activate the Mac Pro's disassembly animation and tuned Telescope's JavaScript time to exclude on-load code and any code after our interaction. We disregarded 400 lines of code before finding the appropriate settings. We found an MVC architecture with event-driven-design, where a `sectionController` and a `clipController` listens for events relative to a `timeline` with functions like `pauseTimeline`, `getVideoHeight`, `resizeFluidAreas`, and `resizeCanvas`. While the clever video playback and container resizing became more evident, we found misleading code that queries and resizes canvas elements when there are none. Similar to the previous two cases, Unravel found hundreds of changes and traces, with the topmost being calls to trigger, enable, and update sections via an `onWheel` handler. Discovering components of the MVC architecture through Unravel was extremely difficult in this case. Unravel provided a sorted view of JavaScript activity based on DOM query count, which highlighted portions of the MVC most active in DOM modification (i.e. pointing to view logic and ignoring model/controller logic).

Runtime Performance

Telescope performed without significant delay on four sites but experienced intermittent frame rate drops on three dynamic sites with hundreds of UI transformations per second. With a relatively small codebase and heavy JS use for SVG modification, the Dot game showed frame-rate drops for 2-3 seconds during some SVG transformations and line renderings. We noticed UI frame rate drops during scroll transitions in 3-4 second intervals for "Snow Fall" as well as some UI delay as all of the startup code traces were transmitted. The Mac Pro website incurred the most significant UI performance delays to less than 1 frame per second for 20 seconds while traces were being processed. In the future, Telescope's performance can be optimized by storing information predetermined about a program's runtime instead of calculating HTML-JS relationships and detail level in real time.

EXPLORATORY USER STUDY

Having demonstrated Telescope's capabilities, we evaluate Telescope's use to answer the following research questions:

RQ2 What programming concepts are users able to elicit using Telescope?

RQ3 What usage strategies do users employ while discovering a web interaction with Telescope?

Method

We conducted an exploratory study with five junior software developers at Northwestern University to understand how they can use Telescope to learn from professional websites. Three of the developers stated they had at least 3 months of professional web development experience through internships. The other 2 stated they knew enough to create website and setup simple JavaScript interactions with libraries like jQuery or Bootstrap. Each user was interviewed about their technical experience and trained to use Telescope for 5 minutes on toy examples. They were then asked to explore 1–3 websites on their own in the time remaining. Sessions lasted 45 minutes each, and each participant was compensated $20. Each participant provided a screen recording with audio for the entire test.

We chose three websites and interactions from the seven in the case study (see Figure 10) that had fun or clever dynamic UI's whose implementation involved at least two functional UI transformations. For each website we observed how users reacted to aspects of code we identified as highly relevant to the UI interaction through prior review. We prompted users to talk-aloud during their interaction and periodically asked them open ended questions such as, "What can you tell me about the way the feature is constructed?", "What coding lessons or decisions can you identify?", and "How does Telescope help in understanding this feature's source?"

Results

In our exploratory study, Telescope helped junior developers quickly identify coding design techniques and programming concepts in the unfamiliar code underlying professional websites, while also inspiring additional discovery. This section addresses our research questions with results from user observations and talk-alouds during user testing.

RQ2: Programming Concepts Recognized

All five users identified front-end software engineering concepts including lazy media loading, mouse position tracking, class-toggled effects, library usage, and animation. A user said, "Seeing what this is helps me know how to approach this problem design-wise (code design)." Four users found an example of lazy-loading and mouse position tracking in XKCD's map viewport by moving the JS timeline constraints past the startup code activity and watching the JavaScript call counts while repeating the map-drag interaction. Two users discovered class-toggled effects by watching the HTML view change during Tumblr's scroll effect, then clicking the HTML query markers to see what JavaScript queried the section element. All users identified instances of library usage in the

Dot game's RaphaelJS line-drawing or each site's jQuery references. Two users found how to construct simple animation through Tumblr's use of jQuery animate.

Seeing in-context front-end architectural patterns working together helped users learn from examples. Users identified patterns for interactive UI including event-driven design, function closures, and state maintenance. Before using Telescope on the XKCD map, a user said, "I know how to make event handlers, queries, and I know the syntax of JavaScript, but I'm missing the *how* of making them work together for a feature like this draggable map." Telescope enabled this user to find multiple patterns in XKCD's comic. Users intuitively found the nature of function closures in JavaScript in scenarios like XKCD's `update` function callback, which contains a `map` variable declared outside the function scope but is referenced without declaration inside the function scope. Users found alternate implementations of state maintenance: storing active state in HTML attributes on Tumblr, or storing the game state in an in-memory JavaScript object via references to `this` in Dot-To-Dot's `Dot` object.

RQ3: Telescope Strategies

We discovered a mix of strategies for interacting with Telescope that our users employed while learning from a UI with Telescope: *constrain-expand*, *copy-paste*, *watch-and-wait*, and *step-constrain-step*.

The *constrain-expand* strategy helped users focus on relevant code and other users curious about library code, external dependencies, or background code. One user said, "The detail control is crazy, because it lets me see just what modified the DOM or I can bring in background code too." *Constrain-expand* was typically used after the user gained a significant understanding of the interaction and wanted to validate their assumptions of hidden variable references or function declarations.

The *copy-paste* strategy emerged when users either tried to play with a portion of code themselves or wanted to see the external media referenced by JavaScript and HTML. The XKCD and Dot-to-Dot websites load external images and audio, which are referenced in Telescope's HTML view. Users copied links to the media to view them as whole files outside the interface. Users copied portions of JavaScript code to an external IDE to see which variables were declared in scope and which ones were not.

We also observed users adopting two other strategies that were less successful in our test. With *watch-and-wait*, users watch the Telescope interface update without adjusting any controls. This made it difficult for users in our test to find interaction code amidst setup code, but could be effective when used on websites with little setup code. Another strategy is *step-constrain-step*, where users narrow the timeline min and max to examine one second of execution at a time. This made it difficult to see calls from high order functions which span multiple seconds, but it was effective in reducing noise from background functions.

Users were able to quickly and easily locate relevant source code for complex interactions. Averaging less than four control toggle changes to find code pertinent to their interaction, users excelled in parsing through fine-tuned views of JavaScript. Three of the users continued exploration past their goal to discover additional coding concepts. One user said, "Once I found that Raphael was being used, I wanted to dig deeper to see how it was configured to make a line wobble."

Developers with less JavaScript experience chose Telescope's HTML pane as a reference point, whereas developers with more experience spent time carefully gaining insights from JavaScript implementation decisions. Telescope's line drawing features helped less experienced developers explore JavaScript from an HTML reference point they felt familiar with. A user said, "This would become my starting point over forums/tutorials – I might even use it on a tutorial's solution instead of reading the tutorial's example code." Telescope's detail expansion feature helped developers with more experience learn architectural decisions about the code. Less experienced developers focused on understanding how to recreate the effects in the default, least detailed view.

DISCUSSION

Having demonstrated the effectiveness of Telescope for helping web developers discover implementations underlying UI interactions, we revisit techniques that contribute to Telescope's effectiveness.

Design Principles for Understanding Unfamiliar Code

The design of the Telescope platform evolved from three prototypes, each shaped by user feedback. Initially we aimed to deliver a code-extracting tool for delivering all code behind an interaction to the users, but providing code by itself was of little value. A participant said, "I can finally see everything that happened, but I don't know what it means." Each subsequent iteration incorporated techniques to present JavaScript and HTML to the user in a way the didn't overwhelm them, which shaped Telescope's three design principles: (1) Bring together relevant JavaScript for an interaction into a single composite JavaScript view. (2) Give the user control over the amount of JavaScript detail they wish to see for any given time frame. (3) Provide affordances to visually link functionality end-to-end, connecting active JavaScript to queried HTML and components in the rendered website. Evaluating the current prototype showed success in helping junior developers understand UI's. All users were able to identify UI engineering concepts in unfamiliar code, and seeing architectural patterns in-context helped users identify how programming techniques can be used to construct a system.

Enabling UI Discovery

Advancing related work [23, 26, 18, 8, 1, 3, 12, 13], Telescope's live tracing and source view constraints helped users identify and understand code supporting an interaction. As a user interacts with a website's UI, Telescope receives trace information and processes it into HTML and JavaScript views for the user. The display of these views are controlled by JavaScript load order, detail, and time constraints. Default settings show the user a focused view of JavaScript responsible for modifying the DOM. Clicking code markers draws lines connecting JavaScript to HTML, helping the user see

how JavaScript manipulates the DOM for a desired outcome. Evaluating the UI discovery in our case study, we found that the source code needed to understand a complex UI behavior is often 150 lines or less.

LIMITATIONS

Instrumentation Scope and Applicability

While Telescope currently supports UI discovery on many popular websites, some limitations prevent it from working on all websites. Scripts that are loaded via lazy-loaders can escape Telescope's instrumentation if they are not present on the page when the Sleight-of-Hand method takes place. Lazy script loaders use URLs in strings to append to scripts to the DOM asynchronously. Telescope will capture and rewrite sources at the time of its invocation, but scripts loaded later are beyond the rewrite scope. However, Telescope does capture calls to load the scripts. Lazy intercepts can be added to Telescope in the future through request blocking and source redirection.

Telescope only instruments and monitors the top-level website frame. Subsequent or nested iFrames were omitted in this project, as iFrames are typically used to embed external content. Future versions of Telescope can recursively traverse the DOM to instrument and listen to traces from iFrames.

While calls to their API's are captured in Telescope, the rendering logic underlying HTML5 Canvas, OpenGl, Flash, Silverlight, and Java Applets are not visible to Telescope. Instrumenting these technologies through website source rewriting is currently not possible.

Performance

Unlike Unravel, Scry, and FireCrystal, Telescope depends on third party servers and lengthy instrumentation processes for large files. The performance overhead required for source instrumentation is considerable on modern hardware and exceeds the capabilities of web browsers. A rich UI might contain fifty thousand lines of code, which can require up to three minutes to instrument. While instrumentated files are cached to speed up repeat-loads, future versions of Telescope could optimize the instrumentation process for larger script transformations by indexing and caching common file subsets like modules and libraries.

Telescope was unable to capture UI interactions on several test sites due to memory limits and website implementation techniques. Telescope sessions for the Netflix and Spotify web players exceeded the browser's memory limitations, resulting in truncated trace data. Amazon's use of iFrames, Airbnb's content security policy, and Forecast.io's app cache script loading prevented Telescope from collecting meaningful trace data. Telescope successfully displays interactions from Google web products, but we found their minification techniques especially difficult to read due to the minification of HTML attributes in addition to JavaScript. In future work, memory problems can be overcome by disabling source tracing and logging for portions of a website until needed, CSP's can be filtered out by debugging proxies, and given enough interest, a crowd of experts could help identify minified HTML attributes.

Code Explanations

Telescope instruments and examines only client-side code and does not curate or explain the code. Further, Telescope does not process or interpret CSS. Existing tools like Theseus and Scry help users discover how server-side code is executed and client side CSS transformations alter the DOM rendering [23, 8]. Future versions of Telescope could incorporate technologies like Tutorons in order to explain the code in the context of active traces [17].

FUTURE WORK

Our future work seeks to transform discovered UI features into portable and indexed deliverables for users to share and learn from, enabling learning communities around real-world examples. Junior web developers struggle with creating UI interactions and experienced web developers have difficulty keeping up with the latest techniques. Enhancing Telescope to support *webstrates* would allow bidirectional modification of a website UI, giving users the opportunity to "sandbox" their UI discovery with a real website [19]. Creating a UI interaction implementation library would help these developers discover techniques used on the web. For example, a user might search for an autocomplete implementation and have the option to compare source code underlying well designed interfaces from Google, Twitter, and Facebook. Further, indexed UI traces from telescope code could be used in-context within IDE's through technologies like Codeletes and BluePrint [25, 5]. With labeling and UI metadata, Telescope's output could be indexed for mining UI *behaviors*, or the combination of user-prompted interaction and underlying source code traces. Output from this mining could be used to elicit implementation patterns or best practices across websites. With a platform that enables example-centric learning from professional websites, we aim to continue lowering the learning barriers present in web development.

ACKNOWLEDGEMENTS

We thank the members of the Design, Technology, and Research program, particularly those in the special interest group Readily Available Learning Experiences (RALE), for helping shape the direction of this work: Jon Rovira, Sarah Lim, Nicole Zhu, Alex Wang, and Christina Kim. We thank Kevin Chen, Henry Spindell, Yongsung Kim, Leesha Maliakal, and Ryan Madden for their design feedback, Darren Gergle, Bryan Pardo, and Aaron Shaw for helpful research discussions.

REFERENCES

1. Alimadadi, S., Sequeira, S., Mesbah, A., and Pattabiraman, K. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, ACM (2014), 367–377.
2. Archibald, J. Deep dive into the murky waters of script loading, 2013.
3. Barton, J. J., and Odvarko, J. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*, ACM (2010), 81–90.

4. Bolin, M. *Closure: The Definitive Guide*. " O'Reilly Media, Inc.", 2010.
5. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2010), 513–522.
6. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2009), 1589–1598.
7. Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, ACM (2013), 473–484.
8. Burg, B., Ko, A. J., and Ernst, M. D. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM (2015), 259–268.
9. Carlini, N., Felt, A. P., and Wagner, D. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012), 97–111.
10. Chang, K. S.-P., and Myers, B. A. Webcrystal: understanding and reusing examples in web authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2012), 3205–3214.
11. Dragicevic, P., Huot, S., and Chevalier, F. Gliimpse: Animating from markup code to rendered documents and vice versa. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, ACM (2011), 257–262.
12. Google. Dev tools tips and tricks, 2016.
13. Google. Inspect and edit pages and styles, 2016.
14. Gross, P., and Kelleher, C. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing 21*, 5 (2010), 263–276.
15. Gross, P., Yang, J., and Kelleher, C. Dinah: An interface to assist non-programmers with selecting program code causing graphical output. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2011), 3397–3400.
16. Guo, P. J. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, ACM (2013), 579–584.
17. Head, A., Appachu, C., Hearst, M. A., and Hartmann, B. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, IEEE (2015), 3–12.
18. Hibschman, J., and Zhang, H. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM (2015), 270–279.
19. Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W., and Beaudouin-Lafon, M. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM (2015), 280–290.
20. Ko, A. J., and Myers, B. A. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2004), 151–158.
21. Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, IEEE (2004), 199–206.
22. Lee, M. J., and Ko, A. J. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, ACM (2011), 109–116.
23. Lieber, T., Brandt, J. R., and Miller, R. C. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, ACM (2014), 2481–2490.
24. Maras, J., Stula, M., Carlson, J., and Crnkovic, I. Identifying code of individual features in client-side web applications. *Software Engineering, IEEE Transactions on 39*, 12 (2013), 1680–1697.
25. Oney, S., and Brandt, J. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2012), 2697–2706.
26. Oney, S., and Myers, B. Firecrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, IEEE (2009), 105–108.
27. Sakamoto, Y., Matsumoto, S., Tokunaga, S., Saiki, S., and Nakamura, M. Empirical study on effects of script minification and http compression for traffic reduction. In *Digital Information, Networking, and Wireless Communications (DINWC), 2015 Third International Conference on*, IEEE (2015), 127–132.

28. Shaffer, D. W., and Resnick, M. " thick" authenticity: New media and authentic learning. *Journal of interactive learning research 10*, 2 (1999), 195–215.

29. Sharp, R. Js bin collaborative javascript debugging, 2016.

30. Souders, S. High-performance web sites. *Communications of the ACM 51*, 12 (2008), 36–41.

31. Victor, B. Learnable programming., 2012.

32. West, M. An introduction to content security policy, 2012.