Automated Recording and Semantics-Aware Replaying of High-Speed Eye Tracking and Interaction Data to Support Cognitive Studies of Software Engineering Tasks

Author 1
Department
University
City
email

Author 2
Department
University
City
email

Author 3
Department
University
City
email

Author 4
Department
University
City
email

Author 5
Department
University
City
email

***Abstract*— The paper introduces a fundamental technological problem with collecting high-speed eye tracking data while studying software engineering tasks in an integrated development environment. The use of eye trackers is quickly becoming an important means to study software developers and how they comprehend source code and locate bugs. High quality eye trackers can record upwards of 120 to 300 gaze points per second. However, it is not possible to map each of these points to a line and column position in a source code file (in the presence of scrolling and file switching) in real time at data rates over 60 gaze points per second without data loss. Unfortunately, higher data rates are more desirable as they allow for finer granularity and more accurate study analyses. To alleviate this technological problem, a novel method for eye tracking data collection is presented. Instead of performing gaze analysis in real time, all telemetry (keystrokes, mouse movements, and eye tracker output) during a study is recorded as it happens. Sessions are then replayed at a much slower speed allowing for ample time to map gaze point positions to the appropriate file, line, and column to perform additional analysis. A description of the method and corresponding tool, Déjà Vu, is presented. An evaluation of the method and tool is conducted using three different eye trackers running at four different speeds (60Hz, 120Hz, 150Hz, and 300 Hz). This timing evaluation is performed in Visual Studio and Eclipse IDEs. Results show that Déjà Vu can playback 100% of the data recordings, correctly mapping the gaze to corresponding elements, making it a well-founded and suitable post processing step for future eye tracking studies in software engineering.**

Keywords*— *Eye Tracking, Empirical Studies, Program Comprehension, High-speed Eye Tracking

I. Introduction

Eye trackers are a critical research tool in understanding how people observe and in turn comprehend visual stimuli [1]. Researchers have successfully used eye tracking hardware to better understand how people read prose, understand diagrams, and process visual landscapes. Computer scientists use eye tracking devices to study how people interact with graphical user interfaces and web pages [2]. The software engineering community is now using eye tracking equipment to study how developers read and understand source code [3]. A detailed and practical guide on conducting eye tracking studies in software engineering is presented in [4].

There are a wide range of eye tracking devices and technology. The devices are made up of hardware, mainly specialized cameras, along with sophisticated software that computes the focal point of the eyes using data collected by the cameras. Additional software is needed to map each of the eye gazes to locations on a visual stimulus (i.e., screen). Eye tracking devices differ greatly with regards to accuracy (of tracking eye movements) and the applications and environments they can be applied to [5]. Studying how people read and comprehend text or source code requires high precision (and costly) eye tracking hardware and software, while determining general spatial regions where a person is looking (left, right, up, down) only requires simple and lower cost hardware and software. Low cost systems cannot identify the exact focus of the eyes, such as what word or letter someone is looking at. They only work well on larger stimuli such as objects in computer games. A high quality, accurate, research-grade eye tracking device allows researchers to determine the exact xy-coordinate on the screen a person is examining. The higher-end eye trackers, in a controlled setting, can pinpoint down to the letter being examined. Research on reading prose and source code does not always require that much accuracy, rather accuracy to the word level is sufficient.

Research grade eye trackers work by presenting an image or text (stimuli) on a computer screen and then using the data from cameras, determine the location (xy-coordinate) the person is looking at. There are a number of limitations to this technology.

The person must be forward looking at the stimuli, cannot look around the room and must be fairly stationary. While these are not serious limitations for conducting scientific studies there is one underlying limitation that poses a substantial road block for studying how programmers understand large, real-world software. Accurate research-grade eye trackers only work on fixed stimuli (i.e., an image or text block) that fits on the computer screen. Changes to the stimuli (screen), such as scrolling or switching files, present a very complex problem. In order to deal with this problem, iTrace was introduced to the research community [6], [7]. iTrace allows a software engineering researcher to conduct eye tracking studies directly in an integrated development environment (IDE) such as Visual Studio or Eclipse. It supports the tracking of eye gaze in the presence of scrolling and context switching. As such, researchers can study developers in a real-world environment and on large realistic software systems. iTrace does this by hooking up to the IDE via a plugin architecture and invoking application and system calls to map the screen xy-coordinate to a line and column in the file in real time. This is then used in a post processing phase to determine the source code token being examined by the study participant.

Eye trackers sample your eyes x-times every second denoted by the frame rate. For example, a 120 Hz eye tracker generates 120 samples per second of raw eye gaze coordinates that needs to be looked up in real time to map to the line, column underlying beneath it. This lookup time is limited to the time it takes for the system calls to return. If the response time of this system call is too long it is not possible to map all gazes coming in accurately to the correct file location. Through use of the iTrace infrastructure we determined that the maximum frame rate at which this can be done in real time is approximately 60Hz (for both Visual Studio and Eclipse). This means that anything above 60Hz will cause the tracker in iTrace to incorrectly map data or drop gaze points altogether. While having a faster computer may help a little, getting to 120Hz, 300Hz or even 1000Hz (at which reading studies are typically done in psychology) is currently impossible with real time mapping.

The work presented here addresses this limitation of the current iTrace architecture by taking all the processing offline. While the IDE API function call response time is fixed, our technique allows for all events to be recorded and replayed back at any given data rate. This allows for mapping gaze data to source code locations with very high-speed eye trackers. The technique is implemented in Déjà Vu, a novel tool that leverages the iTrace infrastructure. The technique and details of Déjà Vu's implementation are presented. The main contributions presented in this paper are:

- **Formalization.** We introduce a fundamental problem in performing eye tracking studies in practical developer environments with high-speed eye trackers.
- **Technique.** We present a novel technique to solve the technological problem presented using automated recording and semantics-aware replaying of eye tracking and interaction data to support cognitive studies of software engineering tasks.
- **Tool.** The novel technique is designed and implemented in a practical tool, Déjà Vu, that leverages the iTrace eye tracking infrastructure.
- **Evaluation.** An evaluation of the fundamental problem with collecting high-speed eye tracking data with and without Déjà Vu is presented in the context of two integrated development environments (Eclipse and Visual Studio) with a sample task.

The paper is organized as follows. Section II presents related work in interaction monitoring. Section III formally presents the problem and motivation for Déjà Vu. Section IV discusses details of the Déjà Vu architecture, design decisions, and how Déjà Vu integrates with iTrace. Section V discusses implementation details of the recording and replaying stages including the challenges faced and how they were mitigated or need managed. Section VI provides an evaluation on the impact of data output rates from eye tracking devices on real-time analysis of eye tracking data on source code with respect to the iTrace framework [6], [7]. Section VII presents conclusions and future work on Déjà Vu's method and implementation.

II. Related Work

Capturing user interaction data for analysis is a common approach in a variety of computational research studies. In [8]–[10] Minelli et al record mouse, keyboard, and IDE interaction data. Fine grain interactions are grouped into broad categories such as comprehension, editing, navigating, etc. to observe developer behavioral during typical tasks. Findings about what activities consume the most developer time, the proportion of development time is dedicated to program comprehension, and the IDE navigational efficiency of developers are presented. The Blackbox project [11] has collected programming interactions within the BlueJ Java IDE for over five years. This dataset has been aimed at providing raw data for research analysis towards better understanding software development behaviors of novice developers. Mylar [12], now known as Mylyn for the Eclipse IDE, allows a developer to track IDE usage activity related to defined tasks. These task contexts can be easily switched in order for developers to multitask without the need to manually relocate artifacts upon returning to a previous task activity. Déjà Vu drastically differs from Mylyn in that Déjà Vu is intended to store interactions along with cognitive information (eye tracking data) for the purpose of replay and subsequent analysis while Mylyn is an active development productivity tool.

ActivitySpace [13] stores mouse and keyboard events related to applications used by software developers to accomplish daily tasks. Event information is logged to a database as an "action record" to create a historical profile of developer interactions. Action records are grouped by a user defined time window and can be queried to help remind developers of resources used and actions taken while working on a given task to improve productivity. Interaction data from ActivitySpace has also been used with machine learning techniques are compared to classify developer activity into higher level categories such as coding, debugging, testing, navigation, web browsing, and documentation [14].

In addition to the capture user interactions, running simulated interactions is a popular solution for software testing research. Sikuli is used in [15] to construct synthetic macro scripts that are application agnostic based on common keyboard and mouse usage. User interactions are supplemented with desktop screenshots and image processing to determine the targets of the actions and automate GUI testing. Specific environments such as websites [16], [17] and Android applications [18], [19] have also been instrumented to record and replay user interactions for the purpose of testing and evaluating web or GUI based applications.

Capture and replay approaches also benefit general purpose automation techniques. The Online Synchronous Education Platform (OSEP) records and abstracts user interactions with websites allowing for editable interactions scripts to be run as pre-recorded or synchronous demonstrations to support educational environments [20]. Using the same framework, an system for automating common or lengthy website interactions is also proposed to improve user productivity [21]. Recent works by Ramler et al. [22] and Bernal-Cárdenas et al [23] take a different approach to capturing and replaying user interaction. Instead of instrumenting applications or recording interactions at an OS level, recorded video of an activity is broken down into individual still frames which are post processed to reverse engineer user interactions shown in the video.

While Déjà Vu makes use of existing recording and replaying techniques, it differs from the state of the art by recreating an eye tracking study in its entirety. User interactions with mouse and keyboard and gaze locations are all replayed to simulate a prior eye tracking study while allowing ample time for more detailed analysis that is not feasible to perform in real-time using high speed eye tracking equipment. Additionally, Déjà Vu affords researchers an opportunity to replicate a study any number of times while analyzing the study in different ways each time to greatly increase the value of participant recording sessions. This is a huge contribution to the current state of the art and provides the eye tracking software research community added incentive to use eye tracking equipment in their studies. The additional advantage of supporting high-speed trackers above 60 Hz (most research grade trackers are 120 Hz or higher) without data loss enables many different types of cognitive analyses (outlined in the next section) that were unable to be done before because of the engineering problem described.

III. Problem Formalization

Eye trackers have been used for decades to study how people comprehend visual stimuli [1]. Modern eye trackers collect a person's eye gaze data on the visual display (referred to as the stimulus) in an unobtrusive way while the subject is performing a given task. This eye movement data can provide very valuable insight into comprehension strategies [24] as to how and why people arrive at a certain solution. Eye movements are essential to cognitive processes because they focus a subject's visual attention to the parts of a visual stimulus that are processed by the brain. Visual attention triggers cognitive processes that are required to perform such things as comprehension. Eye movement is also a proxy for cognitive effort [1] and allows us to determine what parts of a visual stimuli are difficult to understand.

Fig. 1. Gaze plot of a developer's fixations on code.

The underlying basis of an eye tracker is to capture various types of eye movements that occur while humans physically gaze at an object of interest. *Fixations* and *saccades* are the two types of eye movements. A *fixation* is the stabilization of eyes on an object of interest for a certain period of time. *Saccades* are quick movements that move the eyes from one location to the next (i.e., re-fixates). *Dwell time* is defined as the sum of all fixations in a dwell (one visit to an area of interest from entry to exit) [25]. An area of interest is defined by the researcher as any part of the stimulus that is of interest for analysis. For example, in source code, it could be a token or a line. A *scan path* is a directed path formed by saccades between fixations. The general consensus in the eye tracking research community is that the processing of visualized information occurs during fixations, whereas, no such processing occurs during saccades [26]. The visual focus of the eyes on a particular location triggers certain mental processes in order to solve a given task [27]. Modern eye trackers are accurate to 0.5 degrees (0.25 in. dia.) on the screen. In Fig. 1, we see eye gazes on source code (some areas having a much higher density of fixations than others). The fixations are shown as circles on the diagram. The radius of the circle represents the duration of the fixation. The bigger the radius, the more time was spent looking at that particular point. Each fixation has a number displayed in the center of the circle, which indicates the order in which the fixation occurred.

It is important to note that not all eye trackers are made equal. The research-grade eye trackers are thoroughly tested for accuracy quality and reliability compared to low-cost models. Another difference is the frame rate. The low-cost eye trackers capture gazes at a slower rate compared to the research-grade ones. More gazes captured per second give more detailed insight into how people read and analyze software artifacts. Low-cost eye trackers costing a couple of hundred dollars were released for consumer use (mainly gaming). The low-cost eye trackers miss the subtle differences in how humans read and navigate text. The current generation of eye tracking devices offer a wide range of data rates [5]. Older and entry level devices tend to operate at 60 Hz meaning that 60 data points are provided within one second. When performing real-time analysis with received gaze data, analysis tools would be left with approximately 17 milliseconds for any analysis before a subsequent new data point will be received from a tracker. This window narrows as modern trackers are capable of supporting anywhere from 120 Hz to over 2000 Hz.

Eye tracking of source code within an integrated development environment (IDE) is a serious challenge compared to the traditional approach of using static images or text that fit completely on a single screen. In these scenarios, the position of the image or the source text has little or no variance. The gaze data recorded while the stimulus is visible can be mapped down to the pixel on an image-based representation of the data on the display. In an IDE, users may manipulate the view of the source code in any number of ways such as scrolling, file switching, or even editing. These actions require that the gaze data recorded is contextually informed of state of the IDE with respect to the positioning of the source code text and interface elements at a specific moment in time. For example, if a user is scrolling through a source code file looking for a specific identifier, the user's eye positioning may remain fixed within a limited region of the display as the text scrolls past. The issue is that location of the stimulus is changed drastically due to scrolling and it is no longer possible to easily map the screen location of a gaze to the stimulus.

In the case of the iTrace infrastructure [6], [7], where IDE plugins map gaze locations to interface elements and source code text, the combination of latency in plugin environment API calls and the data sampling rate of high-speed trackers significantly limits the feasibility of deep real-time gaze and textual analysis. Currently, solving this problem requires serious tradeoffs. One option is to drop gaze points received while the plugin is busy performing gaze mapping operations causing valuable data points to be lost. Another choice is to buffer all gazes to prevent data loss, but this causes the mapping process to steadily fall behind as the mapping process is a real-time operation and relies on the context of the current state of the IDE when the gaze data is received. This ultimately leads to a desynchronization of the gaze data and the IDE state and renders the data invalid.

Enabling support for high speed trackers allows researchers to collect data for software engineering tasks and better enable them to come to conclusions similar to cognitive psychology reading studies that typically use 1000-2000Hz trackers. We now enumerate several benefits of having support for high-speed trackers implemented in Déjà Vu by extending current eye tracking community infrastructure.

Running realistic studies using the community infrastructure such as iTrace on a tracker greater than 60Hz is now possible as Déjà Vu takes full advantage of the faster frame rate. Most affordable eye trackers are at least 120Hz. This enables researchers to take advantage of the higher frame rate available to them. The higher the sampling rate, the greater the precision of the eye in space and thus there is less error on *dwell time* [25] at any given point on the stimulus. This relates directly to the accuracy of the eye tracker. Accuracy is important when drilling down to specific token the developer is examining. Tokens are of varying length (e.g., short variable names, data types (`int`) or even opening and closing braces) and accurate dwell time is important for a study. With higher precision we can much more accurately map the eyes to the parts of the stimuli with more realistically sized fonts. Currently, to overcome this limitation, researchers use a bigger font, however, this is not very realistic as developers tend to not program with really big. With a 60Hz tracker, the window of error is about 32 *ms* (once every 16 *ms* in either direction) [5].

There are known attentional effects such as attentional cuing [28], inhibition of return [29]–[31], distractor inhibition [32], and flanker effects [33], to name a few, that are highly significant but often quite small and range between 10-15 ms in response and in dwell time. It is impossible to capture these effects with low-precision eye trackers. Many of these effects are highly relevant to software engineering studies. But none of the current studies analyze such effects as there is currently no support to do this in current infrastructure. Note that this is still possible to do with high speed trackers if using short code snippets that fit on the screen, however it has been shown that the results from short snippets do not necessarily translate to realistic tasks [34]. Researchers have studied how eye curvature affects a task. These characteristics can only be discerned at a high sampling rate requiring the use of high-speed tracking. For example, the eye can be attracted to or repelled from a distractor as a function of temporal relationship between a target and a distractor [32]. We have yet to determine if these issues impact real world programming behavior.

Researchers can generally extract a lot more information from high precision data such as pupillary activity [35], [36] and velocity measures that can help with saccade [37] and microsaccade analysis [38]–[40]. *Microsaccades* are miniature eye movements along with tremor and drift that are made during a fixation. They are typically found 1-2 times per second and have an amplitude of between 1’-25’(arcminute). Microsaccades have regained popularity recently and are being studied by eye tracking researchers to learn about the cognitive load [41] and task difficulty [42]. However, to correctly do microsaccade analysis, a 300Hz or higher (500Hz recommended) tracker is necessary to be confident in the velocity measures. Typically, oversampling of the data is used as an alternative but this is not recommended due to the artificial nature of the generated samples. Finally, with the introduction of multiple data collection streams such as studies that incorporate fMRI [43], fNIRS [44], EEG, or GSR with eye tracking, it is recommended to have high speed precision to align timing data.

In summary, we have only begun to start studying developers and cognition in software engineering using eye trackers, however we have yet a lot to learn from cognitive psychology and one of the ways to do this correctly is to have support for high-speed trackers in order to start collecting data correctly and making meaningful conclusions.

IV. The Déjà Vu Approach

As stated previously, calculating all the necessary mappings in real time is not feasible in the context of high-speed eye trackers. To address this problem, it is possible to calculate the mappings after the eye tracking session, as a post-processing step. We record all telemetry data (e.g., keyboard, mouse), along with eye tracker data, and time stamps. This allows us to replay the session in slow motion and calculate mappings as necessary. Hence, we are no longer is constrained by real-time performance requirements.

One method of implementing this is capturing the entire operating system after receiving each gaze during an eye tracking study session. After the study, each operating system state is loaded and all mappings are calculated. This is entirely accurate, however is not practical. It has very poor performance due to requiring copying the entirety of RAM to disk and may require introducing the complexity of a hypervisor. Déjà Vu takes an alternative approach. Only actions that get the environment to each state are recorded and stored. Practically, these are mainly human-computer interaction events – mouse movements and keyboard key state data. Other vital information includes the operating system state history, such as the exact position where a window pops up (in Windows, it depends on where it was previously opened). In these cases, a Déjà Vu style approach needs to take measures to address this and ensure that replays are deterministic. This paper discusses the measures taken by the Déjà Vu tool to address this problem.

In the Déjà Vu approach, the execution process is split into two steps. First, during an eye tracking study, this computer interaction data is collected in real time. After the eye tracking study session is completed, all the computer interactions can be replayed at some later time. This involves replaying the session on the same machine but at a slower frame rate. Since all data is timestamped this can be done without loss and in an accurate manner. Thus, the system/application calls to calculate the line, column in the file can be run without concern and in-depth analysis (of almost any type) can be performed during the replay. Déjà Vu leverages the iTrace infrastructure [6], [7] to capture mouse and keyboard activity during an eye tracking study. To understand the role of Déjà Vu it is necessary to be familiar with the basics of iTrace.

A. iTrace

iTrace is an eye tracking infrastructure to enable research studies within multiple types of software development environments. The infrastructure design is modular featuring three key components, iTrace Core, iTrace Plugins, and an offline post processing application for gaze analysis (see Fig. 2).

The Core provides a unified interface for managing supported eye tracking devices. Through this application eye trackers can be set to calibrate or begin and end eye tracking data recording. All data generated by the eye trackers is first received by the Core which then makes quick decisions based on validity indicators whether the data is acceptable for use by other iTrace infrastructure applications. The Core also provides socket and websocket servers to allow for iTrace Plugins to connect to the Core and receive gaze data for additional processing. In addition to gaze data, the socket communication also coordinates the start and stop of a recording session and subsequent Plugin data processing as well as any output file storage locations for organizational purposes.

Plugins for iTrace support applications such as Eclipse, Visual Studio, and the Google Chrome web browser allow study participants to engage with standard development tools instead of simulated proxies. This allows for data collection to occur in a natural and realistic development environment. Plugins receive the screen coordinate location of a gaze via socket or websocket communication as well as a unique identifier from the Core. Using this information, each plugin performs real-time analysis to map a gaze to contextual information within the IDE or web browsing window. This mapping constitutes line and column positions within a visible source code editing window, IDE interface widgets, or HTML elements (with respect to Google Chrome) that fall under a participant's gaze. These contextual mappings are essential as study participants are free to manipulate the stimulus environment through scrolling, resizing, switching files or pages, searching, and many other activities. Without any kind of context to associate with a gaze, combined with the volatile nature of the stimulus environment, it would be impossible to correctly determine what elements of the stimulus are actually viewed at a given moment in time.

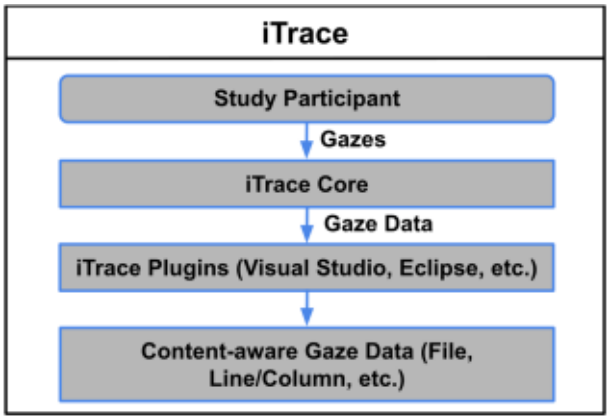

Fig. 2. The architecture of iTrace

All data collected from each eye tracking recording session is stored in XML files. The Core stores participant and study metadata, calibration information, details about the specific tracker used to record the data, and all the raw gaze data points (valid or invalid) received from the eye tracking device during the session. Each plugin records valid gaze points received by the Core and contextual information about the gaze location with the IDE or web browser environment. When a study is complete, the custom offline post processing application provided by the iTrace infrastructure aggregates the data from all XML files. All study metadata and gaze data is collected into a unified Sqlite database where raw gaze data and plugin context information is joined using the aforementioned unique identifiers. Once all of the data is aggregated into the Sqlite database it can be queried using standard SQL commands or further analyzed using the post processing application.

The post processing application provided by the iTrace infrastructure is capable of performing two key analysis methods on the collected study data. The first allows for deeper analysis on all source code context information. Using srcML [45] in conjunction with the line and column information provided by the iTrace IDE Plugins, all textual tokens and the syntactic context of each token within a source code document can be recovered and stored within the database for later querying. Finally, the iTrace post processing application supports three different fixation filtering algorithms (Basic [46], I-VT, and I-DT) each with adjustable parameters. All fixations identified are stored within the database and each fixation references the raw gaze collection that it represents.

The contextual information that iTrace provides is of great value. However, the overhead incurred by collecting this information in real time becomes problematic as the speed at which eye tracking devices are capable of transmitting data increases. To alleviate this issue and fully support high speed eye tracking while still collecting contextual stimulus environment information a new approach is required.

B. Usage Scenario

As far as we are aware, this is the first attempt at supporting high-speed trackers for software engineering-based studies that work on complex artifacts that are tracked within an IDE. We expect Déjà Vu to be used in the following way. A researcher wants to understand how developers understand class hierarchies using a high-speed 1000Hz eye tracker. Before the study, the researcher chooses a suitable real-world code base and the questions a study participant must answer. The code base is imported into a project file in an IDE that has iTrace plugin support (such as Visual Studio). The layout is saved. During the study, a participant is invited in. The eye tracker is calibrated for the participant. The IDE is opened, and the layout is restored. Eye tracking is started in iTrace-Core. Déjà Vu Record is opened, connected to the core, and recording is enabled. At this point, the study participant performs the assigned task. They have the freedom to interact with the IDE, OS, and any applications if they so desire (for example, opening a web browser to access StackOverflow). Once the participant is finished, iTrace-Core and Déjà Vu Record are stopped. The Recording phase is finished. Later, the replay phase begins. Déjà Vu Replay is opened. Analysis plugins are enabled in the IDE and are connected to Déjà Vu Replay. The IDE layout is restored again. Replay is started in Déjà Vu. Everything that happened during the study is now replayed slowly on the computer. Analysis is being performed in the background. Once it is finished, the researcher can collect the data from the plugins and analyze it in any statistical package.

V. Déjà Vu Implementation

Déjà Vu augments iTrace to allow all gaze analysis that occurs in real-time to be deferred to an offline post processing phase. This requires Déjà Vu to record all user interactions. A subsequent replay phase is used to synchronize each user action with respect to recorded gaze data.

A. Recording Stage

During the recording phase (see Fig. 3), Déjà Vu captures human-computer interaction data by recording mouse, and keyboard, along with the eye tracking gaze data. Mouse and keyboard events are captured using Win32 hooks. Hooking into operating system events is a feature of the Windows API and is done through the `SetWindowsHookEx` function. By using this function to hook into low level mouse and keyboard events, Déjà Vu can capture these events before they are added into the input queue. If a study participant is typing code in an IDE, Déjà Vu captures and saves each keystroke before the IDE even receives it. This capturing and saving step happens imperceptibly fast. Performing the capture this way allows for perfect accuracy and replays. Gaze data is collected by listening for broadcasted event data from iTrace-Core.

As this data is collected, it is saved to disk in a CSV format. A sample of the recorded data is shown in Fig. 4. Each row is in the following format: event type, a 64-bit integer specifying the system time, and any data related to the event. This format contains all data necessary for replaying the user's computer interaction. Every event type recorded is shown in Fig. 5 in its CSV format. `KeyDown` and `KeyUp` is used to represent keyboard key state changes. A Windows virtual key code (which is the size of a byte) can store any keyboard key, including modifier keys such as shift or control.

Each of the mouse buttons are explicitly stated as an event type. `Forward` and `back` refers to the buttons on the left side of a mouse (generally used for webpage navigation). `MouseMove` specifies the new absolute position on screen after the mouse has been moved. `MouseWheel` stores any scroll that happens with a value that specifies how much the mouse is scrolled. This event also collects touchpad scrolling on laptops.

The gaze, `session_start`, and `session_end` events are directly retrieved from iTrace Core. Gaze events store the x and y screen coordinate the participant's gaze at that time including validity codes, pupil diameter, and distance to screen. `session_start` and `session_end` events are used by iTrace to mark the beginning and end of a study. These are primarily used to synchronize iTrace Core state with plugins.

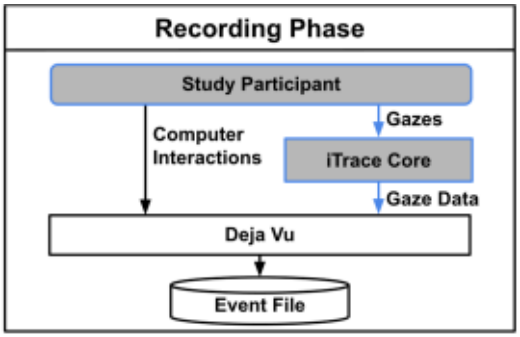

Fig. 3. Recording stage of Déjà Vu. The original steps from iTrace are shaded. The new steps that Déjà Vu adds have white backgrounds.

```
gaze,132277258033906585,314,769
KeyDown,132277258035886613,72
gaze,132277258037224389,336,790
gaze,132277258037601928,333,791
KeyDown,132277258037645064,73
gaze,132277258037758814,323,786
gaze,132277258037914237,333,794
gaze,132277258039069772,270,767
KeyUp,132277258039085245,72
KeyUp,132277258039090178,73
gaze,132277258039225920,276,771
gaze,132277258039755087,316,804
MouseMove,132277258055005185,391,823
MouseMove,132277258055085137,388,823
```

Fig. 4. Example of data collected during the recording phase. Some gazes omitted for brevity.

B. Replaying Stage

During the replaying phase (see Fig 6), Déjà Vu reads in the CSV data produced during the recording stage and replays each event by creating mouse and keyboard events using the

Windows API. Specifically, the `mouse_event` and `keyboard_event` functions are used to synthesize button presses, mouse motions, and mouse scrolls. In addition, Déjà Vu also replays all gazes and emulates the communications protocol used by iTrace Core. This allows existing iTrace plugins to connect to Déjà Vu to receive gaze data and perform analysis during the replay. In essence, Déjà Vu works as proxy for the iTrace Core.

Event Type	Format Description
Keyboard	
KeyDown	Virtual key code
KeyUp	Virtual key code
Mouse	
LeftMouseDown	Mouse button
LeftMouseUp	Mouse button
RightMouseDown	Mouse button
RightMouseUp	Mouse button
MiddleMouseDown	Mouse button
MiddleMouseUp	Mouse button
ForwardMouseDown	Mouse button
ForwardMouseUp	Mouse button
BackMouseDown	Mouse button
BackMouseUp	Mouse button
MouseMove	(x,y) coordinates
MouseWheel	Mouse scroll amount (positive for an upward scroll and negative for a downward)
Eye Tracker	
Raw Gaze – for both left and right eyes	Raw (x,y) coordinates, pupil diameter, validity codes, distance to screen.
Study Session	
session_start	The time when the study session starts.
session_end	The time when the study session ends.

Fig. 5. Timestamped CSV format for all events from the mouse, keyboard, and eye tracker in a study session. Each of the event types is timestamped. The format description includes the main components of each event type.

All events are replayed synchronously. To slow down the replay, Déjà Vu pauses in between events it produces. This pause provides time for connected plugins to process received gaze data. Therefore, time in between events must be carefully considered to give ample time for each connected plugin to perform its analysis. There are multiple possible algorithms for choosing the time to wait in between replaying each event. Déjà Vu implements three such methods so researchers can choose whichever fits their needs the best.

1) Fixed Pause Delay

The time waited after each event is a fixed amount of time based on the type of the event. Plugin processing time for each type of event received will vary depending on the type of analysis performed. Generally, most processing is done after gaze events. Other events, such as mouse movements, may not need any analysis (depending on the researcher's needs). In these cases, processing-heavy events (such as gazes) can be set to have a greater pause time than processing-light events (such as mouse movements).

The primary drawback to this mode is that choosing a good pause length is difficult. Gaze processing latencies are not necessarily easy to predict and outliers are possible. However, via some trial runs a suitable duration could be determined and used. If the experiment is short and fairly simple the fixed paused approach should work well.

2) Proportional Delay

The time after each event is proportional to what it is during the recording. For example, Déjà Vu can set to replay everything at exactly half the speed of recording. This mode is useful for visualizations. Screen recordings performed during the replay stage can easily be sped up by the same factor as the replay is slowed down. Using this method, the sped-up recording of the replay is identical to a recording of the session.

The drawback to this mode is that it is impossible to set a minimum time between events. If processing is to happen after each keypress, nothing stops events from being generated during replay at a very high frequency. During recording, the user could have pressed several keys on the keyboard, generating key presses nearly simultaneously. It is possible that one might want to do some analysis after each keystroke. If the analysis takes 20 *ms*, it would be impossible to set a minimum pause after each keystroke. Even if we slow it down by a factor of 10, if a user pressed 2 keys within less than 2 *ms*, we might not have enough time for analysis. However, this is not an issue for gaze data as eye trackers typically generate readings quite uniformly, making it possible to reinforce a minimum pause time in between gaze events.

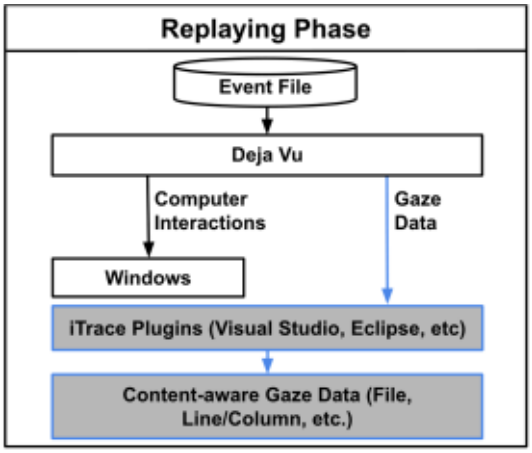

Fig. 6. The Replay stage of Déjà Vu. The original steps from iTrace are shaded. The new steps that Déjà Vu adds have white backgrounds.

3) Bidirectional Delay

In the third method, after gaze events, Déjà Vu waits indefinitely for a reply/acknowledgement from each connected plugin. This reply marks that the plugin is finished doing processing and is ready to process more data. Communication between Déjà Vu and plugins happens bidirectionally. Events that do not need to be waited on are followed by a short fixed-length pause. From a technical point of view, this is the best pausing method. The difficulty of choosing a good fixed-pause length is alleviated. Pauses after gaze events are always correct. No extra time is wasted as padding for the highest-latency lookup/processing cases.

The primary drawback to this method is that it requires modification to the existing components in the iTrace infrastructure. Plugins need to be modified to reply a ready-

signal in response to events that require confirmation. In addition, there is the potential added overhead due to the additional layer of communication that needs to take place.

C. Challenges

While developing the Déjà Vu tool, we ran into several challenges that can be solved in several different ways and will show up in the implementation of tools that use a similar approach to what Déjà Vu does.

1) Solving Non-Deterministic Window Placements

Initial window position is non-deterministic on MS Windows. During a replay, the position where a window opens up can be different from where it opened during recording. To address this, Déjà Vu forces each window opened during replay and recording to open in a single predefined location on the screen. In Déjà Vu, this predefined location is the top left corner of the screen. This is done by frequently iterating over each window handle and checking if any new handles appear.

In theory, this method is not entirely accurate for every application, since the application can move its window without human interaction. However, we have not found a single application that does this to date. To maintain integrity of replays, researchers performing studies need to still consider this issue and avoid using applications in studies that have this behavior.

2) Restoring Initial Interface State

A slight change in interface layouts between runs can cause replay to become out of sync with the events that happen during recording. This can happen in a *butterfly-effect* style. To address this, researchers need to be careful choosing a replicable initial state between runs.

Many IDE's, such as MS Visual Studio, support saving and restoring UI layouts. Saving a layout before running a study and restoring it before performing a replay is a method of ensuring initial interface state in an IDE with adjustable element sizes will remain consistent.

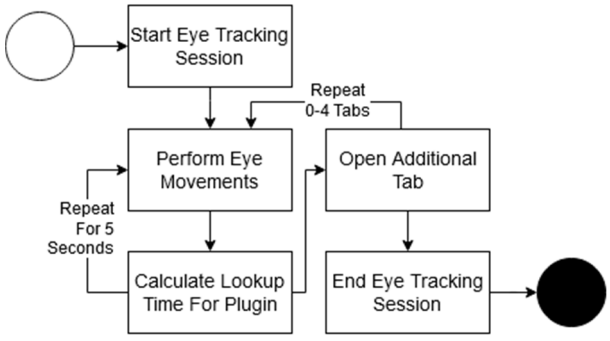

Fig. 7. Process Diagram for Data Collection in Experiment 1

3) Relative or Absolute Mouse Positioning

The MS Windows API allows for two methods of capturing and moving the mouse: by the absolute value (directly specifying mouse position with x and y coordinates) or by relative value (changes the x and y coordinate of the mouse) [47]. Déjà Vu uses absolute mouse values.

The advantage of absolute values over relative value is that replays are more robust. Moving the mouse accidentally during a replay using relative values will cause all subsequent mouse usages to be off. Absolute mouse values solve this issue by automatically locking the mouse back where it should be after each mouse move event.

VI. Evaluation

The evaluation of our approach is conducted via two experiments. Experiment 1 evaluates the initial problem by looking at two typical data analysis plugin implementations (iTrace Visual Studio and iTrace Eclipse) to show data loss and degradation with high-speed trackers. Experiment 2 evaluates Déjà Vu to determine whether it can recreate all gazes that were produced during the recording phase. This is done in the context of a sample eye tracking experiment.

Experiment participants are assigned to one of two groups each denoted by the identifier K and L respectively. Table I shows the eye trackers and data rates used by each group. Each tracker for Group K is connected to a 64-bit MS Windows 10 desktop with a 3.6 GHz Intel i7-7700 CPU, mechanical hard disk drive, 8 GB of RAM, and two 24-inch LCD displays running at a 1920x1200 resolution. Group L eye trackers ran on two separate machines. The machine connected to the Tobii Tx300 used the tracker's built-in 23" monitor running at 1920x1080 resolution on a Windows 10 desktop with 3.5 GHz Intel i7-7800X, a solid-state drive, and 32 GB of RAM. The Gazepoint GP3-HD was connected to a 27" LCD panel running at 1920x1080 resolution, on a Windows 10 laptop with 2.7 GHz Intel i7-6820HQ CPU, a solid-state drive, and 32 GB of RAM.

TABLE I. Participant Groups and the Eye Tracking Devices and Data Rates Utilized

Participant Groups	Eye Tracker Model	Core Data
K	Gazepoint GP3 HD	60Hz
	Tobii Pro X3-120	120Hz
L	Gazepoint GP3-HD	150Hz
	Tobii TX300	300Hz

A. Experiment 1: Data Collection without Déjà Vu

This experiment evaluates the initial problem i.e., does the latency for real time data collection make it infeasible to map eye gaze to semantic elements at high-speed tracking frequencies? To determine this, the iTrace Visual Studio and iTrace Eclipse plugins are evaluated to determine the impact on data rate limitations when performing real-time gaze analysis.

1) Experiment Setup

The iTrace-VisualStudio and iTrace-Eclipse plugin are instrumented to collect timings from the functions related to real-time line, column lookup analysis. The evaluation is run on multiple hardware configurations (Group K and Group L) to provide a less biased performance measure. Each plugin environment (Eclipse and Visual Studio) is also stressed with an increasing number of open source-code tabs to identify potential implementation specific overhead.

2) Data Collection

A process diagram for the first experiment is shown in Fig. 7. An eye tracking study was set up in iTrace. The IDE gaze analysis plugins were connected to iTrace Core. The study participant was instructed to have no files open in the IDE and gaze at the screen for 5 seconds. Then they were instructed to open a file and look at it for 5 seconds. This was repeated until 4 files were opened inside the IDE. Each IDE plugin was modified before the study to collect implementation and environment API performance data. In the iTrace-VisualStudio plugin, this was done using the C# Stopwatch API. Elapsed times for each call to the gaze analysis functionality within the plugin is stored in memory and written out to a file at the end of a recording session. For the iTrace-Eclipse plugin, API performance data was collected using the `System.nanoTime()` API and calculating the difference between the start and stop time for each call to the gaze analysis function. This timing data was stored in memory and written out to a file at the end of a recording session.

3) Results Showing Loss of Data

This data collection process is repeated for each plugin with 0-4 open tabs and the results are presented in Fig. 8. iTrace Eclipse provides an optimized API for translating screen coordinates to the file, line, and column at that screen coordinate. Each lookup in eclipse takes 0.015 seconds. 0.015 seconds is equivalent to approximately 66Hz. This means that real time data collection can only happen for eye trackers operating at 66Hz or less.

Visual Studio does not provide an optimized API for converting screen coordinates to file, line, and column data. For this reason, the lookup timings for iTrace Visual Studio plugin implementation scales linearly with respect to the number of tabs open (due to needing to iterate over all open files). When a single tab is open, the plugin is able to support up to 166hz trackers. However, typically developers have more than a single tab open and any number of tabs open above two will not even support 60Hz. However, both eye tracker speeds estimates are liberal because they do not consider outliers. Fig. 9 shows the raw timing data in the Visual Studio plugin. In conclusion, real time data collection in IDE's using the iTrace eye tracking infrastructure is infeasible for high speed eye trackers (running above 60Hz).

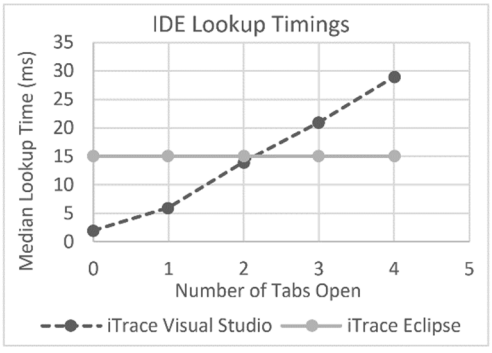

Fig. 8. IDE screen coordinate to file/line/column lookup times in the Visual Studio and Eclipse iTrace Plugins

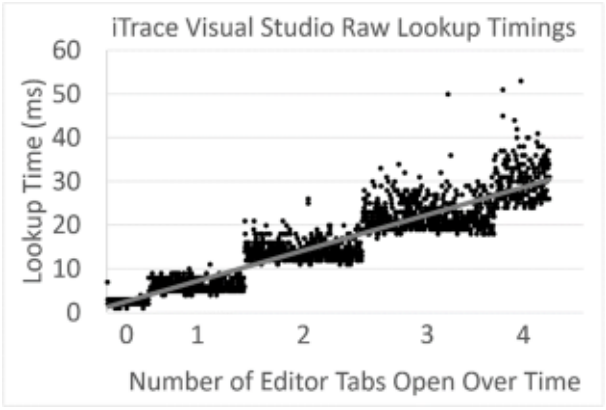

Fig. 9. Raw timing data from Visual Studio. A trendline showing the linear growth is displayed as the number of tabs open increase.

B. Experiment 2: Is Déjà Vu an Effective Solution?

In this section, we describe a simple experiment on two tasks with the goal of showing that Déjà Vu is able to keep up with high speed eye trackers to collect and recreate all gazes that occurred during an experiment.

1) Experiment Setup

The simulated eye tracking experiment consists of two tasks and each task is repeated twice per participant with variations in the data rate of the eye tracking device. The first task requires participants to read out loud each method name and return type from the source code file `SvgExporter.java` taken from the *JHotDraw8* project. This file contains 1,166 lines of code and 42 methods. While this task is straight forward, it will require active engagement with the source code while ensuring a long enough recording duration, minimize cognitive fatigue, and require scrolling.

The second task requires participants to summarize three methods in the `SvgExporter.java` file selected randomly from a collection of the eight largest methods. Participants perform the summarization out loud and the selected methods are not repeated by the participant on the second run of the task when the eye tracker data rate is changed. This task is designed to engage the participant and represent a more advanced eye tracking study task.

2) Data Collection

A process diagram for this experiment is shown in Fig. 10. Participant data captured during the simulated eye tracking study consists of a set of data comprised of: 1) an iTrace-Core data file representing all valid data points generated by the eye tracking device; 2) an iTrace-Eclipse or iTrace-VisualStudio plugin data file containing all data received from iTrace-Core and processed in real-time; and 3) a Deja Vu recording file storing all mouse and keyboard interactions and gaze positions sent from iTrace-Core. Each participant generates two sets of data representing tasks recorded using different eye tracking data rates. Audio recordings of participant activities are also saved via a cellular phone audio recording application. To determine the effectiveness of Déjà Vu's data collection, all gaze data present in the plugin and Déjà Vu output files is compared against the valid raw data points stored and transmitted to each application by iTrace-Core. Gaze data is uniquely identified by

an event id value and is used to determine any data loss (e.g. data transmitted, but not received by the plugin or Déjà Vu).

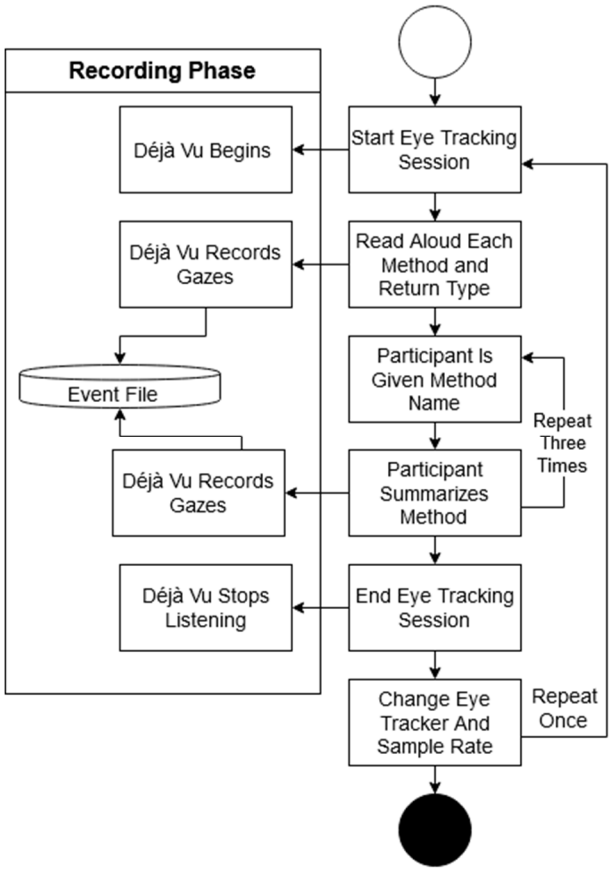

Fig. 10. Process Diagram for Data Collection in Experiment 2

3) Results

Table II shows the data rates of eye tracking devices and the amount of valid data successfully captured by iTrace-Core, Déjà Vu, and the iTrace plugins for Eclipse and Visual Studio. From the table we see that an eye tracking device running at 60 Hz, tends to moderately tax the real-time analysis component of the iTrace plugins. As the data rate doubles to 120 Hz, real-time analysis in the plugins falls behind and nearly half of the data transmitted to the plugins for analysis is lost as plugins cannot keep up with the faster data generate rate of the eye trackers. It is interesting to note that in nearly all cases, the data rate of the eye tracker poses no issue for Déjà Vu with nearly 100% of the data sent from iTrace-Core is also recorded by Déjà Vu along with participant mouse and keyboard interactions.

4) Limitations

We are not implying that the high-speed support for trackers will be needed for every study. Just like not every study needs to be an eye tracking study (there has to be a specific reason for doing that), not every eye tracking study will need to be done using a high-speed tracker. However, as we pointed out in Section III, there are specific use cases for when this is needed. In those cases, Déjà Vu will significantly improve data collection without any data loss or incorrect mapping. Closer investigation of the instances where Déjà Vu did not manage to capture all data points transmitted from iTrace-Core revealed a bug in the research prototype. Occasionally, Déjà Vu can corrupt a data entry which we believe to be caused by a race condition on the output file resource. While this can explain the missing data points given Déjà Vu's generally consistent performance, we still consider these data points to have been lost in Table II to avoid underreporting the findings.

TABLE II. RAW GAZE DATAPOINTS COLLECTED DURING STUDY. THE PERCENT SHOWS THE DATA LOSS. THE K SAMPLES WERE COLLECTED IN THE VISUAL STUDIO PLUGIN. THE L SAMPLES (LAST FOUR) WERE COLLECTED IN THE ECLIPSE PLUGIN.

Sample	Data Rate	Core Data	Déjà Vu	Plugin
K1	60 Hz	22629	22629 (0%)	15817 (30%)
K2	60 Hz	21333	21333 (0%)	19833 (7%)
K3	60 Hz	28392	28392 (0%)	16306 (43%)
K1	120 Hz	41999	41999 (0%)	23424 (44%)
K2	120 Hz	48405	48405 (0%)	26087 (47%)
K3	120 Hz	67024	67023 (<1%)	35786 (47%)
L1	150 Hz	52506	52506 (0%)	25047 (52%)
L2	150 Hz	48090	48088 (<1%)	15858 (67%)
L1	300 Hz	138442	138441 (<1%)	79967 (42%)
L2	300 Hz	106674	106674 (0%)	68852 (35%)

VII. CONCLUSIONS AND FUTURE WORK

The paper presents a unique solution to a fundamental technological problem for studying software developers using high-speed, high-quality eye trackers while working in a realistic development environment on realistically sized software systems. In summary, the Déjà Vu approach captures all relevant user and system interactions for later replay of a user session within a study. The replay allows for very accurate calculation of user gaze points on the entire stimuli of the software system. This overcomes serious real-time limitations posed in mapping screen coordinates to line and column in a given file. The work is aimed at directly facilitating software engineering researchers in studying how developers read software during various tasks (e.g., comprehension, debugging, feature location, etc.). It will also allow the software engineering research community to leverage and apply the more recent advances in cognitive psychology research on text understanding. We believe this will lead to a much deeper understanding of how developers read source code and solve problems which is a complex mixture of many factors. We also foresee this approach being extended to support eye tracking studies in the presence of editing source code. However, this is a very difficult problem and more research is needed to support this type of data collection in an accurate manner.

ACKNOWLEDGMENTS

This work is supported by something. The details are omitted for double blind reviewing.

VIII. References

[1] K. Rayner, "Eye movements in reading and information processing: 20 years of research," *Psychol Bull*, vol. 124, no. 3, pp. 372–422, Nov. 1998, doi: 10.1037/0033-2909.124.3.372.

[2] J. H. Goldberg, M. J. Stimson, M. Lewenstein, N. Scott, and A. M. Wichansky, "Eye tracking in web search tasks: design implications," in *Proceedings of the symposium on Eye tracking research & applications - ETRA '02*, New Orleans, Louisiana, 2002, p. 51, doi: 10.1145/507072.507082.

[3] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A systematic literature review on the usage of eye-tracking in software engineering," *Information and Software Technology*, vol. 67, pp. 79–107, Nov. 2015, doi: 10.1016/j.infsof.2015.06.008.

[4] "Blinded citation."

[5] Holmqvist, Kenneth; Lund University, Nyström, Marcus; Lund University, and Andersson, Richard; Lund University, "Sampling frequency and eye-tracking measures: how speed affects durations, latencies, and more," 2010, doi: 10.16910/JEMR.3.3.6.

[6] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "iTrace: eye tracking infrastructure for development environments," in *10th ACM Symposium on Eye tracking Research and Applications*, Warsaw, Poland, Jun. 2018, p. 3, doi: 10.1145/3204493.3208343.

[7] Bonita Sharif and Jonathan I. Maletic, "iTrace: Overcoming the Limitations of Short Code Examples in Eye Tracking Experiments," presented at the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), Oct. 2016, pp. 647–647, doi: 10.1109/ICSME.2016.61.

[8] R. Minelli, A. Mocci, and M. Lanza, "I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time," in *2015 IEEE 23rd International Conference on Program Comprehension*, Florence, Italy, May 2015, pp. 25–35, doi: 10.1109/ICPC.2015.12.

[9] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying Program Comprehension with Interaction Data," in *2014 14th International Conference on Quality Software*, Oct. 2014, pp. 276–285, doi: 10.1109/QSIC.2014.11.

[10] R. Minelli, A. Mocci, and M. Lanza, "Measuring Navigation Efficiency in the IDE," in *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, Mar. 2016, pp. 1–6, doi: 10.1109/IWESEP.2016.11.

[11] N. C. C. Brown, A. Altadmri, S. Sentance, and M. Kölling, "Blackbox, Five Years On: An Evaluation of a Large-scale Programming Data Collection Project," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, Espoo, Finland, Aug. 2018, pp. 196–204, doi: 10.1145/3230977.3230991.

[12] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, Portland, Oregon, USA, Nov. 2006, pp. 1–11, doi: 10.1145/1181775.1181777.

[13] L. Bao, D. Ye, Z. Xing, X. Xia, and X. Wang, "ActivitySpace: A Remembrance Framework to Support Interapplication Information Needs," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 864–869, doi: 10.1109/ASE.2015.90.

[14] L. Bao, Z. Xing, X. Xia, D. Lo, and A. E. Hassan, "Inference of development activities from interaction with uninstrumented applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1313–1351, Jun. 2018, doi: 10.1007/s10664-017-9547-8.

[15] J. Sun, S. Zhang, S. Huang, and Z. Hui, "Design and Application of a Sikuli Based Capture-Replay Tool," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2018, pp. 42–44, doi: 10.1109/QRS-C.2018.00021.

[16] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive Record/Replay for Web Application Debugging," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2013, pp. 473–484, doi: 10.1145/2501988.2502050.

[17] I. J. Nino, B. de la Ossa, J. A. Gil, J. Sahuquillo, and A. Pont, "CARENA: a tool to capture and replay Web navigation sessions," in *Workshop on End-to-End Monitoring Techniques and Services, 2005.*, May 2005, pp. 127–141, doi: 10.1109/E2EMON.2005.1564474.

[18] F. Yan, M. Xia, Z. Qi, and X. Liu, "Poster: Efficient and Deterministic Replay for Web-Enabled Android Apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 329–330.

[19] J. Guo, S. Li, J.-G. Lou, Z. Yang, and T. Liu, "Sara: Self-Replay Augmented Record and Replay for Android in Industrial Cases," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, 2019, pp. 90–100, doi: 10.1145/3293882.3330557.

[20] Y. Sun, D. Chen, W. Jiao, and G. Huang, "An Online Education Approach Using Web Operation Record and Replay Techniques," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, Jul. 2014, pp. 456–465, doi: 10.1109/COMPSAC.2014.68.

[21] Y. Sun, D. Chen, C. Xin, and W. Jiao, "Automating Repetitive Tasks on Web-Based IDEs via an Editable and Reusable Capture-Replay Technique," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, Jul. 2015, vol. 2, pp. 666–675, doi: 10.1109/COMPSAC.2015.12.

[22] R. Ramler, M. Gattringer, and J. Pichler, "Live Replay of Screen Videos: Automatically Executing Real Applications as Shown in Recordings," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, London, Ontario, Canada, Feb. 2020.

[23] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating Video Recordings of Mobile App Usages into Replayable Scenarios," in *2020 ACM 42nd International Conference on Software Engineering*, Seoul, South Korea, May 2020.

[24] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. 10, pp. 595–609, Sep. 1984.

[25] K. Holmqvist, M. Nyström, R. Andersson, R. Dewhurst, H. Jarodzka, and J. Van de Weijer, *Eye tracking: A comprehensive guide to methods and measures*. .

[26] A. T. Duchowski, "Eye tracking methodology : theory and practice," 2017. http://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=5579221.

[27] M. A. Just and P. Carpenter, "A theory of reading: from eye fixations to comprehension.," *Psychological review*, 1980, doi: 10.1037/0033-295X.87.4.329.

[28] S. Van der Stigchel and J. Theeuwes, "The influence of attending to multiple locations on eye movements," *Vision Research*, vol. 45, no. 15, pp. 1921–1927, Jul. 2005, doi: 10.1016/j.visres.2005.02.002.

[29] M. D. Dodd, S. V. der Stigchel, and A. Hollingworth, "Novelty Is Not Always the Best Policy: Inhibition of Return and Facilitation of Return as a Function of Visual Task," *Psychological Science*, Mar. 2009, Accessed: May 27, 2020. [Online]. Available: https://journals.sagepub.com/doi/10.1111/j.1467-9280.2009.02294.x.

[30] R. M. Klein and W. J. MacInnes, "Inhibition of Return is a Foraging Facilitator in Visual Search," *Psychol Sci*, vol. 10, no. 4, pp. 346–352, Jul. 1999, doi: 10.1111/1467-9280.00166.

[31] J. Lupiáñez, "Inhibition of Return," in *Scholarpedia*, vol. 3, 2010, pp. 17–34.

[32] S. Van der Stigchel and J. Theeuwes, "Our eyes deviate away from a location where a distractor is expected to appear," *Exp Brain Res*, vol. 169, no. 3, p. 338, Nov. 2005, doi: 10.1007/s00221-005-0147-2.

[33] C. W. Eriksen, "The flankers task and response competition: A useful tool for investigating a variety of cognitive problems," *Visual Cognition*, vol. 2, no. 2–3, pp. 101–118, Jun. 1995, doi: 10.1080/13506289508401726.

[34] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic, "Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters," in *Proceedings of the 41st International Conference on Software Engineering*, Montreal, Quebec, Canada, May 2019, pp. 384–395, doi: 10.1109/ICSE.2019.00052.

[35] A. T. Duchowski, K. Krejtz, N. A. Gehrer, T. Bafna, and P. Bækgaard, "The Low/High Index of Pupillary Activity," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, Honolulu HI USA, Apr. 2020, pp. 1–12, doi: 10.1145/3313831.3376394.

[36] A. T. Duchowski *et al.*, "The Index of Pupillary Activity: Measuring Cognitive Load *vis-à-vis* Task Difficulty with Pupil Oscillation," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, Montreal QC, Canada, 2018, pp. 1–13, doi: 10.1145/3173574.3173856.

[37] S. Van der Stigchel, M. Mills, and M. D. Dodd, “Shift and deviate: Saccades reveal that shifts of covert attention evoked by trained spatial stimuli are obligatory,” *Atten Percept Psychophys*, vol. 72, no. 5, pp. 1244–1250, Jul. 2010, doi: 10.3758/APP.72.5.1244.

[38] R. Engbert and R. Kliegl, “Microsaccades uncover the orientation of covert attention,” *Vision Research*, vol. 43, no. 9, pp. 1035–1045, Apr. 2003, doi: 10.1016/S0042-6989(03)00084-1.

[39] Z. M. Hafed and J. J. Clark, “Microsaccades as an overt measure of covert attention shifts,” *Vision Research*, vol. 42, no. 22, pp. 2533–2545, Oct. 2002, doi: 10.1016/S0042-6989(02)00263-8.

[40] E. Lowet, B. Gomes, K. Srinivasan, H. Zhou, R. J. Schafer, and R. Desimone, “Enhanced Neural Processing by Covert Attention only during Microsaccades Directed toward the Attended Stimulus,” *Neuron*, vol. 99, no. 1, pp. 207-214.e3, Jul. 2018, doi: 10.1016/j.neuron.2018.05.041.

[41] C. Kelleher and W. Hnin, “Predicting Cognitive Load in Future Code Puzzles,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI ’19*, Glasgow, Scotland Uk, 2019, pp. 1–12, doi: 10.1145/3290605.3300487.

[42] A. Duchowski, K. Krejtz, J. Zurawska, and D. House, “Using Microsaccades to Estimate Task Difficulty During Visual Search of Layered Surfaces,” *IEEE Trans. Visual. Comput. Graphics*, pp. 1–1, 2019, doi: 10.1109/TVCG.2019.2901881.

[43] B. Floyd, T. Santander, and W. Weimer, “Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, May 2017, pp. 175–186, doi: 10.1109/ICSE.2017.24.

[44] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, “The effect of poor source code lexicon and readability on developers’ cognitive load,” in *Proceedings of the 26th Conference on Program Comprehension - ICPC ’18*, Gothenburg, Sweden, 2018, pp. 286–296, doi: 10.1145/3196321.3196347.

[45] M. L. Collard, M. J. Decker, and J. I. Maletic, “srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration,” in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, Sep. 2013, pp. 516–519, doi: 10.1109/ICSM.2013.85.

[46] P. Olsson, “Real-time and Offline Filters for Eye Tracking,” Masters Thesis, KTH Electrical Engineering, Stockholm, Sweden, 2007.

[47] Microsoft, “mouse_event function (winuser.h) - Win32 apps,” *mouse_event function*, Dec. 05, 2018. https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-mouse_event (accessed Feb. 24, 2020).