gazel: Supporting Source Code Edits in Eye-Tracking Studies

Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland
Washington State University, USA
{first.last}@wsu.edu

Cole S. Peterson
University of Nebraska-Lincoln, USA
Cole.Scott.Peterson@huskers.unl.edu

Venera Arnaoudova
Washington State University, USA
venera.arnaoudova@wsu.edu

Bonita Sharif
University of Nebraska-Lincoln, USA
bsharif@unl.edu

Jonathan I. Maletic
Kent State University, USA
jmaletic@kent.edu

***Abstract*—Eye tracking tools are used in software engineering research to study various software development activities. However, a major limitation of these tools is their inability to track gaze data for activities that involve source code editing. We present a novel solution to support eye tracking experiments for tasks involving source code edits as an extension of the `iTrace` [9] community infrastructure. We introduce the `iTrace-Atom` plugin and `gazel` [gəˈzel]—a Python data processing pipeline that maps gaze information to changing source code elements and provides researchers with a way to query this dynamic data. `iTrace-Atom` is evaluated via a series of simulations and is over 99% accurate at high eye-tracking speeds of over 1,000Hz. iTrace and `gazel` completely revolutionize the way eye tracking studies are conducted in realistic settings with the presence of scrolling, context switching, and now editing. This opens the doors to support many day-to-day software engineering tasks such as bug fixing, adding new features, and refactoring.**

I. Introduction

Eye tracking tools and techniques are increasingly being used in software engineering research to study participants interactions with source code. Traditional approaches trace gazes over static images of code, where the text does not move relative to the screen for the duration of an experiment. The static nature of images prevents researchers from analyzing data from experiments conducted in realistic settings, where the text visible on the screen can be dynamically changed via scrolling, editing, or switching between multiple files.

To address this problem, the `iTrace` [9] infrastructure was developed to map gaze locations to specific source elements within an IDE. An eye tracker provides an `(x,y)` coordinate indicating where a gaze is located relative to the active display, and the IDE translates this `(x,y)` coordinate to a row and column in a source code file within the editor. From this row and column information, researchers use parsers to identify the source code element that corresponds to the row and column being viewed.

This process assumes that the source code file being parsed is static, i.e., no edits are made during the experiment. Edit support has been a widespread limitation for gaze mapping tools, and greatly impacts the types of software engineering tasks that can be studied using eye trackers (e.g., bug fix or feature addition).

This paper presents a novel solution to address the editing limitation, as an extension to the `iTrace` [9] infrastructure. We propose `iTrace-Atom`, a plugin that tracks gaze and edit information over source code files in the Atom editor, accompanied by `gazel` [gəˈzel] (gaze edit evolution) a Python data processing library to analyze the data collected by `iTrace-Atom`.

Researchers can use these tools to track source code elements as they move and change throughout the course of an experiment, all while maintaining accurate gaze fixation information. For example, researchers can track how long participants looked at a source code element of interest throughout a task, without losing valuable gaze data when an element is deleted, edited, or moved.

The paper provides the following contributions to the research community.

- **Source Code Editing Support.** To enable researchers to study a variety of software tasks that involve source code editing during eye tracking experiments, we present a novel technique to capture source code edits and map eye gaze data to evolving source code.
- **Tracking Source Code Evolution.** We present a novel technique to track the evolution of source code elements, at identifier-level granularity, over the course of an experiment, including elements that are moved, renamed, or refactored.
- **Extended Language Support.** We present a novel data processing pipeline, `gazel` , in the form of an extensible Python library. The library includes parsers for all mainstream languages, improving upon existing support with 23+ new languages.
- **Support for high-frequency eye trackers.** Eye tracking studies performed with existing plugins can drop up to 60% of gaze information from high frequency eye trackers due to high latency from IDE plugin environment. `iTrace-Atom` can capture 99% of all gaze information from eye trackers with data sampling rates up to 2,000Hz

important for in-depth cognitive analyses such as microsaccades [8].

II. Approach

The steps described in Sections II-B and II-C are implemented as part of the `iTrace-Atom` plugin which is published to atom.io and can installed via apm[1]. The steps described in Sections II-D–II-G are implemented as a collection of functions part of `gazel` which can be installed via pypi[2]. Documentation, detailed usage instructions, and demonstration video can be found in our online replication package [5].

A. Overview of the Approach

A high level overview of the approach is shown in Figure 1. First, all edit actions performed by a user during an experiment are captured by `iTrace-Atom` and saved to a change log. Throughout the experiment, the initial version of any source code file opened in the IDE is saved to memory. After an experiment is completed, `gazel` parses all source code files and uses the edit information saved in the change log to recreate different versions of the file in the form of source code snapshots. Snapshots represent the version of a source file at different points in time during the experiment.

All gazes captured by the eye tracker and resolved to line and column values by `iTrace-Atom` are saved to a file, which `gazel` then partitions. Gazes that occurred during the time for which a specific snapshot is valid will be paired to that snapshot file. Gazes are then processed using a fixation filter.

All snapshot files will now have gaze data mapped to the correct syntactic tokens. However, in order to understand how gaze and code edit information changes across snapshots, `gazel` uses the change log to create an in-memory object to represent the source code evolution during an entire experiment. Tokens are linked across snapshots, changes to a token's content or location are resolved by time, and fixation information is mapped to tokens in the appropriate snapshot.

B. Recording Gaze Data

Gaze data is captured by the eye trackers and streamed to `iTrace-Atom` via `iTrace Core` [9]. Gaze data is in the form of `(x,y)` screen coordinates, which `iTrace-Atom` must resolve to line and column numbers.

To do this, we first scale the `(x,y)` coordinates by the scale factor of the primary active display. Any coordinates outside of the active editor window bounds are recorded as invalid gazes. We then calculate the bounds of the source code within the active file, by taking into account the file gutter, the folded code, and any active toolbars which may offset the file location. We then pass the final coordinates to Atom's `getLine` and `getColumn` functions, to obtain the appropriate line and column location given an `(x,y)` coordinate. This data is written asynchronously to an .xml 'Gaze' file, which contains data for each gaze including: `(x,y)` coordinates, line and column information, timestamp, and the active file name.

[1]https://atom.io/packages/itrace-atom

[2]https://pypi.org/project/gazel/

C. Capturing Edit Information

Edit information is captured directly from the Atom IDE. Once `iTrace-Atom` is in the tracking phase, a change log JSON file is created containing the following information: *File:* The file which was edited. *Type:* The type of edit. Edits can be an insert or a delete. For example, a cut-and-paste action would be registered by the IDE as a delete and then an insert. *Offset:* an integer denoting the position of a character in the text buffer at which the edit was made. *Text:* The contents of the edit, i.e., the inserted or deleted text. This is always a single character if the text is inserted or deleted character by character. *Length:* The length of the text. This is used to determine how the character offset buffer has been modified. *Timestamp:* The time of the edit in Unix epoch time. *Row, Column:* The starting row and column numbers in which the edit was made.

D. Creating Source Code Snapshots

Challenge: In order to generate correct gaze information, eye tracking data provided from the eye tracker at time t_n needs to be mapped to the appropriate source code elements for the snapshot of the code that was being viewed at the same time t_n. All versions of the source code need to be captured and parsed in order to generate correct gaze information.
Solution: At the start of the experiment, the original version of the source code files open in the editor are saved. Once tracking starts, all edit information is captured in a change log throughout the experiment. The original version of the source code file is parsed using `TreeSitter` [6], a parser generator tool that builds editable syntax trees. Edits from the change log are sequentially applied to update the syntax tree. Each updated version of the syntax tree is considered as a snapshot of the original file, at a certain point in time. Consecutive edits, which are made close in time (3 seconds by default; customisable), are aggregated and applied together, to make a single snapshot with multiple changes to the syntax tree.

Evaluation: To ensure correctness of the reproduction of edits in `gazel` several different scenarios have been thoroughly tested, including simple edits, multi-line deletes, copy-pasting, replacing of text, multi-location edits (using the multiple cursors and find/replace), and changes introduced by Atom's autocomplete functionality. To verify the correctness of snapshot reproduction `gazel` compares the final generated snapshot to the final file version saved by Atom plugin.

E. Partitioning the Gaze Set

Each snapshot represents the state of the source code file at a certain point in time, between two aggregate edit actions. All gaze information that is captured between two edit actions can now be accurately mapped to source code elements in the corresponding file snapshot. Because the IDE resolves line and column information in real time, all line and column values recorded in the gaze file are correct. The appropriate version of the code that a user was viewing is associated with the gazes captured for that time period.

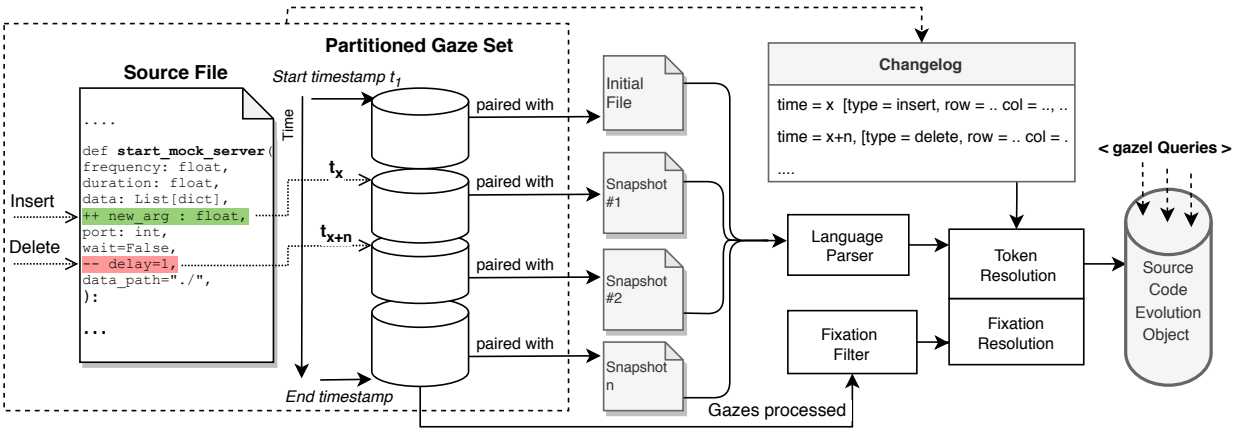

Fig. 1. gazel: overview of the approach and data processing pipeline.

F. Gaze Fixation Filters

`gazel` provides the option of processing raw gaze data saved by the Atom plugin using various fixation filters to generate fixations from the combined raw gaze files tagged with AST information. Currently, `gazel` provides the same fixation filters from the `iTrace Toolkit` [2]. The algorithm parameters are customizable, and options to provide alternative fixation algorithms are supported.

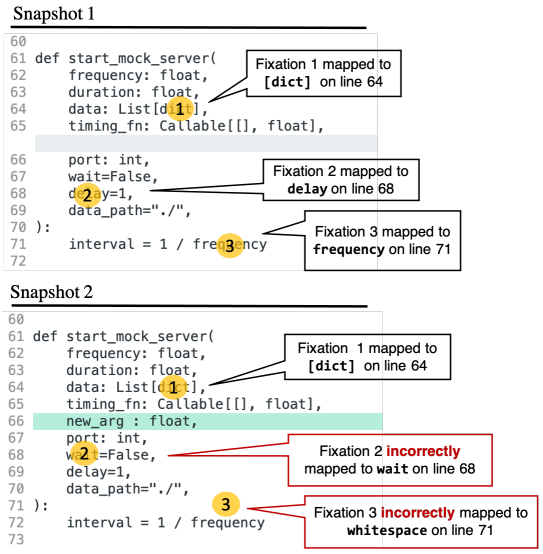

Fig. 2. Challenges of tracking gazes on edited source code.

G. Tracking Changes Across Snapshots

Challenge: Tracking how gazes change as syntactic elements are edited, inserted, or deleted from a file is a major challenge. Figure 2 contains two snapshots of a source code file, `snapshot 1` is the original file and `snapshot 2` has an new line inserted on line 66. Three gaze fixations recorded on `snapshot 1` are displayed as yellow circles on the figure. Fixation 1 is recorded on line 64, fixation 2 on line 68, and fixation 3 on line 71. Current solutions assume the text at a specific line and column is static, and resolve syntactic tokens by parsing the source file. However, when a new line is inserted on line 66, fixation 2 and 3 are still mapped to lines 68 and 71. However, the syntactic token under the fixation has changed. Fixation 3 was correctly mapped on to the identifier 'frequency,' but with the edit, is now mapped to white space. A viable solution needs to keep track of all edits, and resolve gazes the correct syntactic tokens.

Solution: `gazel` provides functionality to track both source code elements and gazes across edits. `gazel` goes over every edit in the change log, and detects which tokens and gazes existing before the edit are affected by it. This allows tracking a token from the original version of the file to the final version with a record of how it changes, along with any fixations associated with it.

This is implemented in a two-step approach: First, for the original version of the source code, we construct a parse tree in which we assign a unique id to every token; we update fixation data with the source code token information as well as the ids. Second, for every edit, we apply it and we parse the source code. We then use `Tree-Sitter` to obtain all the tokens in the parse tree that semantically changed from the previous version of the parse tree. If a token has not changed semantically, we assign it the same id that it had in the previous version of the source. Otherwise, the token is assigned a new unique id. We then update fixation data with the source code token information as well as the ids.

At the end of this process, each source code element in each version of the file will be tagged with an id. This id is stable through time; if two source code elements have the same id, they are guaranteed to be the same source code element regardless of time or source file version. It is important to note here that fixation data is always correct for the snapshot of the file at the time it was recorded, and hence can be reliably attributed to a source code token if applicable. Thus, we only need to track source code elements across edits, and that information can then be used to recover the gazes on a specific

source code element from previous snapshots. `gazel` provides a high level API to allow the use this information to perform aggregate analysis. For example, it allows users to obtain all fixations over a single or a set of source code elements across multiple edits, get fixation data with adjusted position information for a given snapshot (see Figure 2), and identify what tokens changed due to a particular edit, and how it affected their associated gazes.

H. Supporting High Speed Eye Trackers

Challenge: Existing eye tracking plugins struggle with the high latency of IDE API calls, which prevents real time gaze and textual analysis at the data sampling rate of high-speed eye trackers as gazes are lost due to the high latency. One way that researchers have addressed this problem is to move any real-time analysis into an offline tool (e.g., `Déjà Vu` [12]). However, software engineering researchers often require quick analysis of eye tracking data for use in post-experiment walk-through of the data to gain more insight into the gathered data for which real-time resolution of `(x,y)` coordinates is needed.

Solution: In order to support real-time resolution of `(x,y)` coordinates, we create `iTrace-Atom`—an open-source cross platform text editor. Atom exposes APIs to efficiently map gaze data to line and columns, and is extremely fast, taking less than $1ms$ to resolve `(x,y)` coordinates to line and column in the text editor (compared to existing plugins for Visual Studio where a single API call takes $15ms$). To ensure no gazes are dropped, we limit the number of API calls needed at the eye tracking data rate by caching editor information, and writing data asynchronously.

Evaluation: To evaluate the efficiency of `iTrace-Atom`, we run an experiment by generating mock eye tracking data within the editor window at rates from 60Hz–2,000Hz for variable lengths of experiment time, and number of files open in the editor. We compare the number of gazes sent by the script to the number of gazes received and resolved to line and column numbers by the plugin. For data rates 60Hz–120Hz Atom is able to capture 100% of gazes. For 150Hz–2,000Hz Atom's data retention drops slightly but remains higher than 99.7%. Number of files and experiment time has no measurable effect. Compared to existing plugins this is a major improvement. `iTrace-VisualStudio` plugin captures 64% of gazes at 60Hz and sharply drops to 6% and 3% for rates of 1,000Hz and 2,000Hz. This decreases even more when multiple files are open in the editor [12]. `iTrace-Eclipse` plugin performs similarly to `iTrace-Atom` up to 300Hz, but then can only capture 30–15% of gazes at 1,000–2,000Hz. Experiment data and scripts can be found in our replication package [5].

III. Usage scenarios

Through the support of source code edits, `iTrace-Atom` and `gazel` allow researchers to conduct a much wider variety of eye tracking studies including:

1) Conducting studies involving a large variety of software maintenance tasks that involve source code editing (e.g., feature implementation and testing).
2) Analyzing the evolution of source code elements throughout the course of an experiment.
3) Maintaining access to gaze areas of interest (AOIs) that are connected to a single source code element, even if it is modified or shifts location within a file.

IV. Related Work

The software engineering research community has made significant effort to develop the tools and infrastructure needed to make experiments with source code and eye tracking devices feasible, with plugins designed for popular IDEs such as Visual Studio and Eclipse [9], [12]. Researchers have also developed essential tools to help with eye tracking data processing [1], [3], [4] and visualization of eye-tracking data (such as fixations and gaze paths) over source code elements [7], [10], [11]. However, to the best of our knowledge, `gazel` is the first to support source code editing actions and to resolve eye-tracking data over evolving source code. `gazel` empowers researchers to conduct software engineering experiments with a larger spectrum of tasks where participants can edit source code, and provides a simple way for researchers to query complex source code evolution information for data analysis.

V. Limitations and Future Directions

Limitations: `iTrace-Atom` does not track gazes that occur at the same time as an edit action, as they can not be accurately paired to one source code snapshot. Certain IDE features, such as split code windows, are not supported, and a detailed list of limitations can be found in `iTrace-Atom`'s documentation [5]. *Future directions:* In the future, we plan to add support to track refactoring edit actions across files. Moreover, to ensure the usefulness and usability of `gazel` , we plan to conduct user experience experiments with participants, and reach out to researchers to assess their needs and the usefulness of the data processing pipeline in `gazel` . Finally, we plan to extend edit support to the existing `iTrace` plugins for Visual Studio and Eclipse in the near future, integrating it into the `iTrace` infrastructure [9].

References

[1] Eye Code. http://github.com/synesthesiam/eyecode.
[2] iTrace-Toolkit. http://www.i-trace.org/toolkit_doc_home_0-1-0/.
[3] Pandas Eye. http://github.com/hanav/PandasEye. Accessed: 2020-11.
[4] PyGaze. http://www.pygaze.org/. Accessed: 2020-11.
[5] Replication-package. https://github.com/Smfakhoury/gazel.
[6] Tree-Sitter. https://tree-sitter.github.io/tree-sitter/.
[7] B. Clark and B. Sharif. iTraceVis: Visualizing eye movement data within eclipse. In *VISSOFT*, pages 22–32, 2017.
[8] R. Engbert and R. Kliegl. Microsaccades uncover the orientation of covert attention. *Vision research*, 43(9):1035–1045, 2003.
[9] D. Guarnera, C. Bryant, A. Mishra, J. Maletic, and B. Sharif. iTrace: Eye tracking infrastructure for development environments. In *ETRA*, pages 1–3, 2018.
[10] N. Peitek, S. Apel, A. Brechmann, C. Parnin, and J. Siegmund. CodersMUSE: multi-modal data exploration of program-comprehension experiments. In *ICPC*, pages 126–129, 2019.
[11] D. Roy, S. Fakhoury, and V. Arnaoudova. VITALSE: visualizing eye tracking and biometric data. In *ICSE: Companion*, pages 57–60, 2020.
[12] V. Zyrianov, D. Guarnera, C. Peterson, C. Scott, B. Sharif, and J. Maletic. Automated recording and semantics-aware replaying of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks. In *ICSME*, 2020.