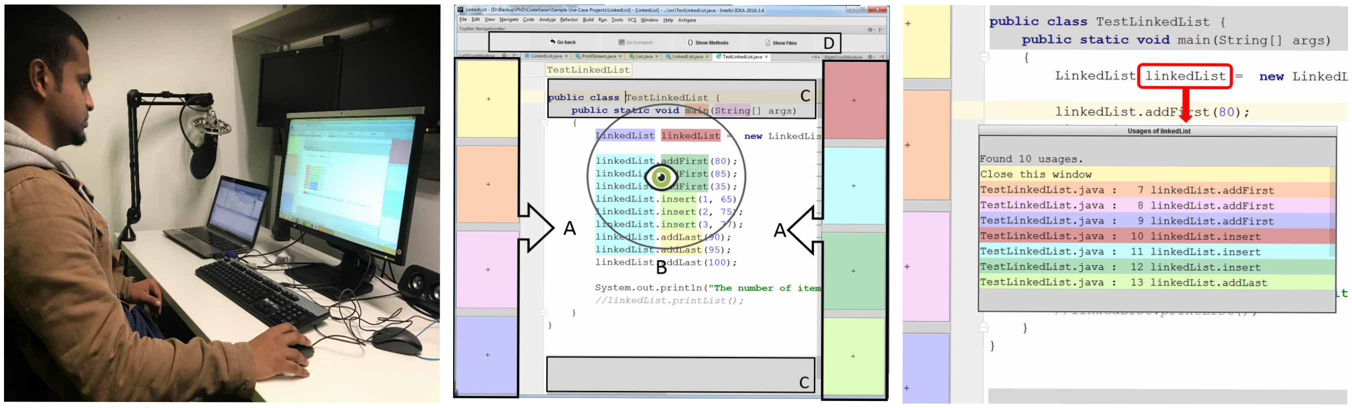
CodeGazer: Making Code Navigation Easy and Natural with Gaze Input

Asma Shakil
Media Design School,
University of Auckland
Auckland, New Zealand
asma.shakil@mediadesignschool.com

Christof Lutteroth
University of Bath
Bath, UK
University of Auckland
Auckland, New Zealand
c.lutteroth@bath.ac.uk

Gerald Weber
University of Auckland
Auckland, New Zealand
g.weber@auckland.ac.nz


LinkedList - [D:\Backup\PhD\CodeGazer\Sample Use-Case Projects\LinkedList] - [LinkedList] - ...\src\TestLinkedList.java - IntelliJ IDEA 2016.3.4
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help Actigaze
TopBar NavigationBar
Go back
Go Forward
Show Methods
Show Files
D
LinkedList.java
PrintStream.java
List.java
LinkedList.java
TestLinkedList.java
TestLinkedList
C
A
A
B
C
Usages of linkedList

```
public class TestLinkedList {
    public static void main(String[] args)
    {
        LinkedList linkedList =  new LinkedLis

        linkedList.addFirst(80);
        linkedList.addFirst(85);
        linkedList.addFirst(35);
        linkedList.insert(1, 65);
        linkedList.insert(2, 75);
        linkedList.insert(3, 77);
        linkedList.addLast(90);
        linkedList.addLast(95);
        linkedList.addLast(100);

        System.out.println("The number of item
        //linkedList.printList();
    }
}
```

```
Found 10 usages.
Close this window
TestLinkedList.java :   7 linkedList.addFirst
TestLinkedList.java :   8 linkedList.addFirst
TestLinkedList.java :   9 linkedList.addFirst
TestLinkedList.java :  10 linkedList.insert
TestLinkedList.java :  11 linkedList.insert
TestLinkedList.java :  12 linkedList.insert
TestLinkedList.java :  13 linkedList.addLast
```

Figure 1: Left: Experimental setup. Middle: CodeGazer interface showing confirm buttons in the margin (A), highlighted identifiers in the region of the gaze (B), gaze-aware scroll bars in the editor (C), and the top navigation bar (D). Right: Gaze-responsive modal usage window.

ABSTRACT

Navigating source code, an activity common in software development, is time consuming and in need of improvement. We present CodeGazer, a prototype for source code navigation using eye gaze for common navigation functions. These functions include actions such as "Go to Definition" and "Find All Usages" of an identifier, navigate to files and methods, move back and forth between visited points in code and scrolling. We present user study results showing that many users liked and even preferred the gaze-based navigation, in particular the "Go to Definition" function. Gaze-based navigation is also holding up well in completion time when compared to traditional methods. We discuss how eye gaze can be integrated into traditional mouse & keyboard applications in order to make "look up" tasks more natural.

CCS CONCEPTS

• **Human-centered computing → Pointing**; • **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

Eye gaze tracking; Source code navigation; Integrated Development Environment (IDE); Actigaze

ACM Reference Format:
Asma Shakil, Christof Lutteroth, and Gerald Weber. 2019. CodeGazer: Making Code Navigation Easy and Natural with Gaze Input. In *CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019), May 4–9, 2019, Glasgow, Scotland UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3290605.3300306

1 INTRODUCTION

Gaze tracking has the potential to become ubiquitous since popular device designs have user-facing cameras and in the future may provide integrated gaze tracking [5, 6, 9, 29, 44, 46]. Using gaze for interaction is an important research area because it provides instantaneous benefit to the user, whereas in other applications such as gaze analytics the user chiefly has a passive role. Much research has focused on gaze-only alternatives for clicking [10, 12, 15, 30, 40, 45]. We posit that the broadest benefit can be provided if gaze can be used in synergy with other interaction techniques and employed in everyday tasks. In this paper we use source code navigation tasks as an interesting case study. Our results indicate that gaze can hold up against mouse in the workplace, even if the mouse sits right beside the user.

Source code browsing and navigation are two tasks commonly performed in software development, especially during source code maintenance and evolution [4, 20, 22]. It has been reported that 60-90% of software development costs involve reading and navigation of code as part of programmers' maintenance tasks [7].

Software developers mostly work with an Integrated Development Environment (IDE) to ease the process of writing, compiling, searching and debugging code. IDEs facilitate source code browsing by providing search and navigation features such as the ability to "Find All Usages" or "Go to the Definition" of an identifier. These search actions are typically provided via context menus attached to an identifier in code. They are usually also accessible to the user via shortcut keys on the keyboard and/or via direct clicking with the mouse.

In spite of the features provided by IDEs, literature shows that programmers encounter significant navigation overhead when searching through code. In a study of experienced Java developers, Ko et al. [22] found that developers using the Eclipse IDE, spent on average 35% of their time with the mechanics of navigation, going back and forth between related code. Piorkwiski et al. [42] found that during debugging tasks, participants spent on average 50% of their time foraging for information in code.

In this work we wanted to investigate whether proven gaze interaction techniques can be employed to tackle the above research challenges. We have developed a plugin for the IntelliJ IDEA IDE [18] called CodeGazer. This allows users to perform common navigation actions such as "Go to Definition" of an identifier, "Find All Usages", navigate to files and methods, move back and forth between visited points in code and scroll – all through gaze input. We use the Actigaze input method, which was developed in our working group. Actigaze is a gaze-only click alternative which has been shown to be both fast and accurate in clicking small targets such as hyperlinks in a web page [30, 39]. Actigaze has also been used in other projects [23]. Here we are investigating Actigaze in combination with traditional input methods such as mouse and keyboard.

In particular, we aimed to answer three research questions:

RQ1 How does gaze-based code navigation compare with keyboard-based and mouse-based navigation in terms of performance, accuracy and usability?
RQ2 Will users choose gaze-based navigation if they have a choice of input modality?
RQ3 Which navigation features can be best supported with gaze interaction?

To address these questions, we conducted two studies. The first study, addressing RQ1 and described in Section 4, was a *comparative study* which compared the efficiency and accuracy of gaze-based navigation with that of keyboard-based and mouse-based navigation. The second study (see Section 5), addressing RQ2, was a *free-use study* which was aimed at assessing user preference. Both studies address RQ3.

Our work makes several key contributions: first, we describe how Actigaze, a proven gaze-based interaction paradigm, can be applied for code navigation, discussing several nontrivial challenges in the coloring algorithm. Secondly, we provide a working prototype, CodeGazer, as a plugin to the IntelliJ IDEA IDE for doing gaze-based navigation in Java code. Thirdly, we present the results of an empirical evaluation which shows that:

- Gaze-based navigation is comparable to keyboard navigation in speed but slower than the mouse.
- Programmers made few mistakes (6.9%) when using gaze input to navigate code.
- Programmers mostly chose gaze input in a voluntary usage choice.
- Gaze input is most suitable for the "Go to Definition" action.

With the advent of eye gaze tracking technology in mainstream computing devices [13], we consider that a tool such as CodeGazer may give software developers an option to browse code more quickly and easily.

2 BACKGROUND AND RELATED WORK

Understanding Programmer Navigation

A large body of research work has been directed towards understanding how programmers navigate source code. Ko et al. [22] found that developers spent, on average, 35% of their time with the mechanics of navigation within and between source files while working on Java code in the Eclipse IDE. They also found that, overall, an average of 27% of the developers' navigations were those going back and forth between related code. They suggest that tools be developed to replace the context menus which IDEs provide for inspecting code with something requiring fewer steps. In another study which was aimed at eliciting design requirements for maintenance-oriented IDEs, Ko et al. [20] found that "some tools in the Eclipse IDE seemed to slow programmers' progress by imposing significant interactive overhead (e.g. extra clicking and visual searches)" (p. 128). Our work addresses this problem by trying to minimize the interaction overhead during code navigation and, by using an alternative interaction technique, offering more choice.

Murphy et al. [35], reported on usage data collected from 41 Java developers working in the industry and using the Eclipse IDE. They used the Mylar plugin, which they developed to record the interaction history of all users. They found that 74% of the developers used the Package Explorer view of the Eclipse IDE to help locate a point of interest, while only 12% (five out of 41) used the Eclipse-provided Bookmarks to navigate between points of interest in the code.

Minelli et al. [33] studied 740 development sessions of 18 developers and collected data amounting to about 200 hours of development time. They found that unsurprisingly, users spend a large proportion of their time trying to understand code (70%). A rather worrisome finding was that developers spend roughly 14% of their time in fiddling with the user interface of the IDE. They suggest that "there is a need for research on novel user interface and interaction paradigms for the IDE, a still unexplored field with some exceptions" (p. 33).

A number of researchers have looked at explaining code navigation behavior with Information Foraging theory (IFT). In one of the early studies on this topic [28], researchers looked at the relationship between programmers' hypotheses about the program's behavior and their navigation strategies. Piorkwiski et al. [41] designed recommender tools based on programmers' "scent" to aid them during the task of debugging. They evaluated their effectiveness in an empirical study, showing that participants found IFT-based recommendations very useful. A related study by the same group [42] concluded by saying that programming tools sometimes may not help programmers as they put too much demand on users' attention to be useful.

Gaze as Input for Code Navigation

There have been only a few attempts at using eye gaze interaction as an input mechanism in a programming environment. Glucker et al. [8] have enhanced an IDE prototype with gaze-controlled interaction methods for source code navigation. Their system named EyeDE comes closest to ours in terms of intent as it also tries to use gaze to provide hands-free navigation for reading and understanding code. It allows users to perform several navigation functions using gaze, such as opening files, looking up a method's documentation, navigating between its declaration and implementation, and navigating to a method using a code outline. The main mechanism used for gaze interaction is direct dwell, e.g. onto a source code symbol such as a method name, to open a context menu and then on a context menu item to trigger an action. While this appears to be fairly intuitive, it suffers from the so-called "Midas Touch" problem [16, 17], a well-known fundamental problem in using gaze for selection tasks: there is a risk of triggering "inadvertent clicks" when the user's gaze is misinterpreted. Furthermore, EyeDE is affected by tracking inaccuracy: source code symbols are usually too small to be distinguished with gaze pointing alone if some of them are close together [48]. In Section 3, we explain how CodeGazer overcomes Midas touch and tracking inaccuracy by using Actigaze [30] for the selection of source code symbols.

In more recent work, Radevski et al. [43] used a combination of gaze and keyboard shortcuts to allow users to navigate source code without the mouse. Their system named EyeNav allows users to position the cursor, scroll in the direction of gaze, move the caret horizontally along a line or vertically along a column in code, do single character movement, and select text in the editor. EyeNav relies heavily on keyboard input and allows users to move the cursor with the keyboard to compensate for gaze tracking inaccuracy. By contrast, CodeGazer is gaze-only and focuses on navigating the logical structure of code.

Gaze as Input for Other Domains

Gaze input has been used for other interaction tasks such as typing [24, 27, 31, 32, 34], pointing and selection [11, 14, 37] and general-purpose interaction [25]. In order to improve speed and mitigate Midas touch for selection with gaze, researchers have proposed dynamically adjusting the dwell time of targets. The adjustments are based on the likelihood of the targets' selection [34] and the eye movement time estimated by Fitts' law [14]. Other work combined gaze with traditional input for pointing, application switching and scrolling [25, 26]. Our work uses a variant of 'smooth scrolling with gaze-repositioning' [26] in order to support gaze-based scrolling in the code editor and the modal windows.

3 CODEGAZER DESIGN

Gaze-Based Click Interaction

CodeGazer supports gaze-based navigation for the features listed in Table 1, using the Actigaze concept to click targets [30, 39]. Clickable targets in CodeGazer include identifiers in the source code, navigation buttons in a top bar and list items in a modal window (see Figure 1).

Actigaze employs a two-step procedure to accurately select a discrete target with gaze. When users look a target they want to click, clickable items within a predefined radius of the gaze point are highlighted with colors (in our study the eight closest items within 5cm). Each distinct clickable is highlighted with a unique color and associated with a correspondingly colored confirm button. There are eight stable

Table 1: Navigation features supported by CodeGazer.

Navigation Feature	Gaze Input	Keyboard Input	Mouse Input
Go to Definition Find All Usages	Dwell on identifier in code Dwell on confirm button	Place caret on identifier Press Ctrl+Alt+g	Place pointer on identifier Press Alt+Left Button
List All Files List All Methods	Dwell on the top navigation bar Dwell on confirm button	Ctrl+Alt+0 (Files) Ctrl+Alt+9 (Methods)	Click “Show Files” button Click “Show Methods” button
Go to a Usage Go to a File Go to a Method	Dwell on usage in modal window Dwell on confirm button	Arrow keys to navigate in modal window Enter to select	Mouse pointer to point in modal window Left Click to select
Go Back Go Forward	Dwell on the top navigation bar Dwell on confirm button	Ctrl+Alt+1 (Go Back) Ctrl+Alt+2 (Go Forward)	Click “Go Back” button Click “Go Forward” button

confirm buttons placed in the margins of the window (see Figure 1), with colors that are easily discerned. Users then click the desired target by dwelling (in our study for 300ms) on the correspondingly colored confirm button.

The use of confirm buttons in gaze-based clicking offers a two-fold advantage. First, it helps to clarify the target which the user is intending to click, even though it may be too small to target accurately using direct gaze pointing. This is particularly important for clicking identifiers for code navigation because identifiers are often short (e.g. variable names) and often appear close together (e.g. in formulas). Direct gaze pointing would lead to many “incorrect clicks” [48]. The second advantage of confirm buttons is that they help to avoid the Midas touch problem: they are in the periphery, away from the normal source code content, so users are unlikely to look at them unless they want to click. Also, since the confirm buttons are at stable positions in the IDE, users subconsciously memorize their positions very quickly, making it quick and easy to locate them on screen [30]. The confirm buttons have a cross hair in the middle as a visual anchor and to provide visual feedback for a click [40].

Highlighting Identifiers in Source Code

The algorithm used to highlight identifiers in CodeGazer is a variant of that used to highlight hyperlinks in Actigaze [30]. On static web pages, Actigaze uses a stable coloring algorithm. A link is assigned a color when the page is loaded and the color remains unchanged until a new page is loaded. For this color assignment Actigaze uses the spatial layout of hyperlinks on the web page to minimize the likelihood of close occurrences of the same color. Each clickable is assigned the color for which all clickables of the same color are spatially furthest away from it. CodeGazer faces a color assignment problem which is more general for two reasons. The first is that source code is editable in nature (as opposed to the mainly static nature of web pages) and the spatial relationships between the clickables change as code is added or deleted in the editor. Secondly, unlike web pages where each link maps to a unique target, multiple usages of an identifier in code all map to the same target. These multiple usages of an identifier are scattered throughout the code; assigning colors to them based on a local layout may result in color clashes in another region of code, with different identifiers within proximity of each other getting the same color. CodeGazer thus does not assign a color to the identifiers on loading a new file. Instead, the color assignment is re-evaluated on each dwell event in the editor.

The color assignment algorithm needs to satisfy three requirements. First, there should not be more than eight distinctly colored items at any instant in the text editor. This is because there are only eight confirm buttons in the user interface (Figure 1) and each colored identifier must have a one-to-one correspondence with a confirm button. So, in each dwell event, the algorithm chooses to highlight only those eight items that are closest to the gaze point. Secondly, all usages of the same identifier should be highlighted with the same color (Figure 1) since they map to the same navigation target. The coloring algorithm considers the semantics of highlighted identifiers and assigns the same color to all identifiers that resolve to the same reference. Thirdly, the color of an identifier should remain stable (i.e. not flicker due to a constantly changing color assignment) as long as the identifier is highlighted. To achieve this stability of colors, CodeGazer maintains a record of the colors assigned to the identifiers in the current dwell event. If these identifiers are within the gaze radius in a subsequent dwell within the editor, then the colors last assigned to them are used again. Only those identifiers which were not highlighted previously but are now within the gaze radius are then assigned one of the eight possible unassigned colors.

Navigating in the Code

When a user dwells on a particular identifier in code, an association is built between all currently highlighted identifiers in the code and the correspondingly colored confirm buttons in the margin. If the user now dwells on an associated confirm button, this triggers one of two possible navigation actions: either "Go to the Definition" of the identifier or "Find All its Usages." The actual action which is triggered depends on whether the confirm button is associated with a definition or with a usage in the code. To make this distinction between the definition of an identifier and its usage, CodeGazer assigns different colors to these two different types of occurrences of an identifier. Clicking a confirm button which is associated with the usage of an identifier takes the user to the definition of that identifier and places the caret at the location of the definition. This is similar to how the "Go to Definition" option works in an IDE. On the other hand, if the user dwells on a confirm button which is associated with the definition of an identifier, it opens a modal window which lists all usages of that identifier (see Figure 1). This action is the same as the "Find All Usages" option in an IDE. The modal window is gaze responsive and users can scroll in it with gaze. The usages in the window are again highlighted with colors, and a particular usage is selected by dwelling on the corresponding colored confirm button (Figure 1 middle).

To navigate to any file in the project or to any method in the current file or move back and forward in code, users need to first dwell on the top navigation bar (see Figure 1). This assigns a color to each button in the navigation bar. To click the button, users again need to look at the corresponding colored button in the margin. Clicking on the "Show Files" or "Show Methods" button brings up a modal window listing all files or methods as the case may be. Users can then choose to open any file or navigate to any method by using the same controls as those used for navigating to a usage in the "Find All Usages" navigation action (see Table 1).

4 COMPARATIVE STUDY

In this study we examined performance and accuracy in the context of three input conditions – keyboard, mouse and gaze. Keyboard-based and mouse-based conditions served as a control for evaluating gaze-based input. The independent variable in our study was the interaction technique used for navigation. The dependent variables measured were "time taken to complete a navigation task" (for performance), "number of incorrect navigations" (for efficiency) and "SUS score" (for usability).

Methodology

To measure the motor time of navigation actions or errors in navigation, we could ask participants to do individual navigation actions such as go to definitions of an identifier or look up its usages in code or navigate to methods or files. However, such individual navigation actions done in isolation lack realism. This is against recommended practice for empirical evaluation of development tools [36].

Study Design. We designed study tasks based on actual debugging scenarios. However, participants were told the exact navigation steps which they needed to do to uncover the bug. This was done to eliminate program comprehension time and different navigation actions across the three input conditions. We designed the mouse and keyboard conditions to remove confounding factors (e.g. visual layout) and change only the input modality. This made them visually as consistent as possible while keeping the overall interaction the same as originally in the IDE (see Table 1).

Code Base and Tasks. We used three different tasks during the study. Task 1 was based on code which had two classes and 92 lines and was used to add elements to a custom array class. The bug which was seeded was that one of the three insert functions did not increment the size of the array. To uncover the bug, participants needed to perform six navigation steps (which comprised three "Go to Definition" and three "Go Back" actions). Task 2 was based on code with six classes and 276 lines and simulated a student enrollment system. The seeded bug was a missing break statement in a switch clause which resulted in wrong course enrollments for Year 1 Software Engineering students. To uncover the bug, the participants had to perform nine navigation steps (which comprised three "Go to Definition," three "Find All Usages" and three "Go to Usage" actions). Task 3 was based on code which had nine classes and 2141 lines and simulated a library management system. The seeded bug was in a function which returned book details in an array which was populated from index 1 instead of index 0. Participants needed to perform three navigation steps to uncover the bug (which comprised three "Go to Definition" actions).

Procedure. The participants had been emailed an overview of the study prior to the session. The procedure started with the participants filling out a consent form. They were then told to seat themselves comfortably in a typical wheeled office chair and adjust its height if needed. The eye tracker device was calibrated for them once at the beginning. Each task followed the same sequence of execution. The purpose of the task was explained and the bug was shown to the participants by running the code. The participants did one round of practice for the navigation steps which they were told to do for the task at hand before the measurements were taken. The participants repeated the navigation steps for gaze, keyboard and mouse input. To avoid bias due to learning effects, the order of input condition was permuted across the three tasks. Once the participants had completed all three tasks, they were asked to fill out a demographics questionnaire. The participants were also emailed a System Usability Scale (SUS) questionnaire at the end of the study which they could fill out at their convenience.

Participants. A total of 24 participants (five females) took part in our comparative study. The participants' ages ranged from 19 to 42 years (M = 23.1; SD = 5.2; Mdn = 21). Seven of the participants had prior experience using an eye tracker, 14 participants wore glasses and two were color blind. Participants reported coding on average for 3.17 hours a day. Furthermore, two of the participants were lecturers in Software Engineering, one was a tutor in Software Engineering, four were undergraduate students in Software Engineering and the remaining 17 were postgraduate students of either Computer Science or Software Engineering.

Results

We collected a total of 1296 navigation actions from 24 participants, i.e., 432 navigation steps for each interaction technique across the three tasks. To compare the three interaction techniques, we measured the task completion time as well as the number of incorrect navigation actions for each condition and each participant.

Task Completion Time. The frequency distributions of the mean task completion times for the three interaction techniques are shown in Figure 2 and the boxplots of mean task completion times are shown in Figure 3. A Repeated Measures ANOVA was conducted to compare the effect of interaction technique on mean task completion time. The interaction technique had a significant effect on task completion time (F(2,46) = 25.55, p < 0.001). Post-hoc comparisons with Holm correction showed that there was no significant difference in completion time for gaze (M=20198, SD=4051) and keyboard (M=20661, SD=4329), t(23)=-0.44, p = 0.664. However there were significant differences in the completion times between gaze and mouse (M=14805, SD=2923), t(23)=5.77, p < 0.001, and between keyboard and mouse, t(23)=8.19, p < 0.001. These results suggest that while the mean task completion time of gaze is comparable to that of keyboard, it is significantly higher than that of the mouse (see Figure 5a), i.e. gaze is slower than the mouse.

Taskwise Analysis. We also compared the task completion time of individual tasks across the three interaction techniques. A Repeated Measures ANOVA showed a significant effect of the interaction technique on Task 1 completion time (F(2,44) = 7.69, p = 0.001), on Task 2 completion time (with

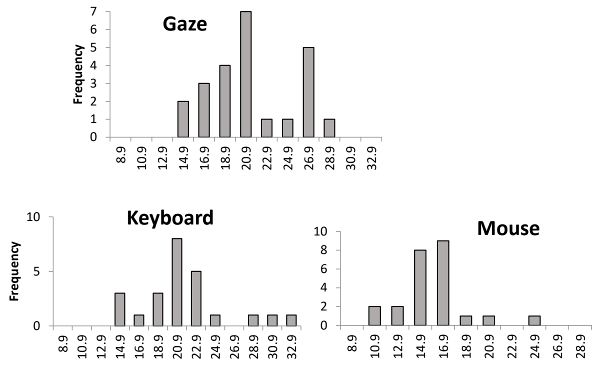


Figure 2: Frequency distributions of task completion times (in seconds) in the comparative study.

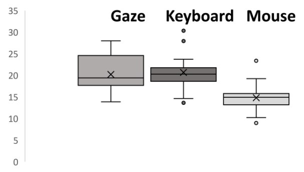


Figure 3: Boxplots of task completion times (in seconds) in the comparative study.

Huynh-Feldt correction) (F(1.4,31.5) = 21.87, p < 0.001) and Task 3 completion time (F(2,46) = 20.18, p < 0.001). The individual task completion time results are shown in Figure 5b.

Accuracy. The frequency distributions of the errors made across all tasks for the three interaction techniques are shown in Figure 4. The error counts have a non-normal distribution, therefore non-parametric statistical tests were used to analyze the data. A Friedman rank sum test showed a significant effect of the interaction technique on the accuracy, $\chi^2(2, N = 24) = 11.76$, p = 0.003. Post hoc comparisons with Holm correction showed significant difference between gaze and keyboard, t(46)=3.35, p = 0.005, and between gaze and mouse, t(46)=3.35, p = 0.005. There was no significant difference between keyboard and mouse, t(46)= 0.0, p = 1.0. These results show that there are significantly more errors with gaze input than with keyboard or mouse.

System Usability Score. The System Usability Score (SUS) of CodeGazer (M = 76.42, SD = 10.92) as translated to an adjective rating scale was "Good" [1]. SUS scores lie between 0 to 100, with a higher score indicating better usability. An SUS score of 70 or above shows that the system is at least passable while systems with scores above 90 are considered highly usable [2].

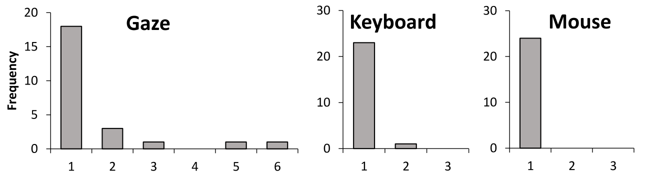
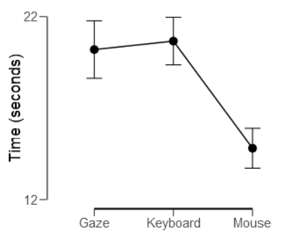
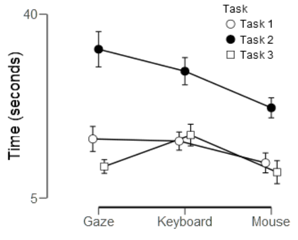


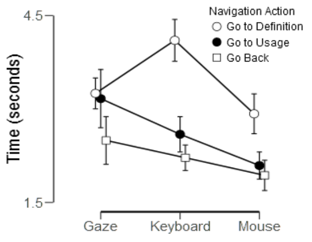
Figure 4: Frequency distributions of navigation errors in the comparative study.



(a) Mean Completion Time



(b) Taskwise Comparison



(c) Featurewise Comparison

Figure 5: Analysis of completion times in the comparative study.

Discussion

In the comparative study our goal was to evaluate the feasibility of gaze input by comparing it to keyboard and mouse input (RQ1). Our results show that the mean task completion time of gaze input is slightly better than the keyboard, although it is significantly worse than that of the mouse (see Figure 5b). It is hard to beat the mouse as the established gold standard for mouse-only direct clicking tasks.

The taskwise analysis results show that in Task 3 gaze performed significantly faster than keyboard and very close to the mouse (Figure 5b). Task 3 only included “Go to Definition” navigation actions. This result suggests that the “Go to Definition” feature performs particularly well with the gaze input (RQ3). Our analysis of the effect of input device on the timing of individual navigation actions confirms this conclusion (Figure 5c). Gaze performed significantly faster than keyboard and close to the mouse for the “Go to Definition” feature. This may be due to the fact that this feature requires users to just *look at* the corresponding colored button in the margin. A corresponding action with the mouse would require the users to place the mouse pointer on the identifier in the code and then press the Alt+Left Mouse button (equivalent to Ctrl+Left Click in the default IntelliJ implementation). With the keyboard, it would require users to first place the caret at the identifier in code and then press the required shortcut key (Ctrl+Alt+g). This is cumbersome when compared to simply looking at a confirm button. The mouse performs much better than gaze for tasks that require mouse button clicks only (without key presses), including actions such as choosing usages in a modal window or clicking the “back” button in the top navigation bar.

5 FREE-USE STUDY

In this study we assessed whether users would use gaze input if they had a choice of interaction technique (RQ2). We measured the frequency of use for keyboard, mouse or gaze input during navigation.

Methodology

We wanted to use a methodology which would let participants do navigation actions in their most *natural* style without being constrained by the demands of the study design. In the previous comparative mode of study, although the tasks were based on actual debugging scenarios, there was no need for participants to engage with the code as they were given the exact navigation steps to perform. While this structured mode of navigation is essential for establishing time and error metrics, it makes the tasks artificial and we feel it may not be sufficient to evaluate user preference even subjectively. Other researchers have tried other techniques for assessing preference which go beyond the usual Likert-scale or ranking approach most commonly used in the literature. Besancon et al. [3], for example, asked participants at the end of their study to pick a favourite input technique (between mouse, tactile and tangible input) and perform 15 docking tasks. The experiment had already taken over an hour, so if participants were asked to perform an additional set of trials, they would have a strong incentive to pick the solution they really preferred. Voluntary usage is a good metric for evaluating user preference; it has been pointed out in literature that “data showing voluntary usage is really the ultimate subjective satisfaction rating” [38].

Study Design. As in the comparative mode of study, we again asked participants to perform debugging tasks. However, this time they were not told the steps needed to complete the tasks. Participants were only shown a bug by running the code and were given a starting point at which to begin navigation. They were then free to choose any input technique they wanted at each step of the navigation process. This mode of study thus provides a more immersive experience for the participants by requiring them to read through the code, understand its structure and logical flow and navigate in the code as they see fit. To mitigate novelty and social desirability bias when choosing the preferred technique, we used several strategies. First, participants had the opportunity in the pre-task phase to play with the gaze option. Secondly, we conducted a detailed interview with the participants at the end of the study, asking them about specific reasons they chose a particular navigation technique and about their general experience of using CodeGazer. Thirdly, we encouraged participants to provide genuine feedback, emphasizing that there is no 'right choice' and that we were genuinely interested in knowing which of the methods worked best for them.

Codebase and Tasks. We used all three tasks from the comparative study (see Section 4). We also added a fourth task, which used the same codebase as Task 2 of the comparative study but with a different bug. The seeded bug was located in the method to generate the course code for Math Year 2 students, which resulted in incorrect course codes; the method swapped the use of the course number and year, both of which were passed in as parameters.

Procedure. The procedure followed the same style as the comparative study. Participants were emailed an overview of the study prior to the study session. Participants were comfortably seated and performed a gaze calibration process. We first ensured that participants were properly familiarised with all navigation features (gaze, keyboard and mouse). Then tasks were shown to them by running the code and pointing out the bug. They were given the starting point in the code from where they should begin navigation. Also, they were specifically told that they could choose any interaction technique they wanted (gaze, keyboard or mouse) during the course of finding the bug. The task was finished when the participant reached the location of the bug in code. During the task, participants were allowed to ask questions from the moderator, if they needed help. This process was repeated for all the four tasks in the study. Once the participants had finished the tasks, a detailed interview was conducted with questions pertaining to their experience with CodeGazer and their specific reasons for choosing particular input techniques. A demographics questionnaire was administered at the end of the study session to avoid effects of *stereotype threat* on participants' performance [21, 47]. The complete procedure took about an hour.

Participants. A total of 28 participants (five females) took part in our free-use study. The participants' ages ranged from 19 to 35 years (M = 22.0, SD = 2.9, Mdn = 21). Nine of the participants had prior experience using an eye tracker, 19 participants wore glasses and none were color blind. Participants reported coding on average for 2.9 hours a day. One of the participants was a tutor in Software Engineering, two were undergraduate students in Software Engineering, and the remaining 25 were postgraduate students of either Computer Science or Software Engineering. Eighteen of the participants had also taken part in the comparative study of CodeGazer.

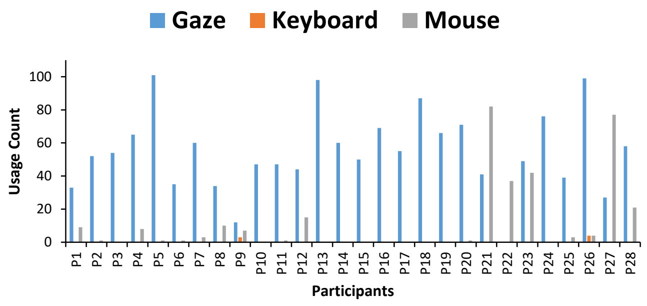


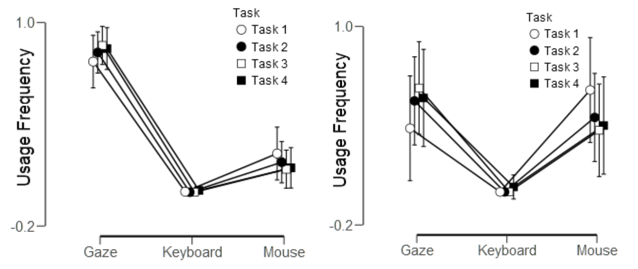
Figure 6: Usage count per participant in the free-use study.

Results

We collected a total of 1859 navigation actions across all 28 participants and all four tasks in the free-use study mode. There was considerable variation in the participants' overall completion times (M = 821.49, SD = 297.42, 95% CI [931.66, 711.33]) as well as the total number of navigation actions they performed (M = 66.39, SD = 25.02, 95% CI [75.66, 57.13]).

Usage Frequency. The usage frequency of each input modality is shown in Figure 6. It shows that 21 of the 28 participants used gaze more than 80% of the time during the study, with ten of them using it 100% of the time and nine others using it more than 90% of the time. Only three participants used the mouse more than 70% of the time. Further, only two participants ever used the keyboard, and that less than 7% of the time. There was no decline in gaze usage over time. Most participants kept using gaze throughout the one-hour study session and across all the four tasks. A comparison of those participants who used mixed mode interaction is shown in Figure 7b. We did not observe an effect of programming experience on the choice of input modality.

A Repeated Measures ANOVA with Huynh-Feldt correction (since Mauchly's test indicated a violation of sphericity) showed a significant effect of the interaction technique on usage preference (F(1.14,33.043) = 64.49, p < 0.001). Post-hoc comparisons with Holm correction show that there is a significant difference in usage preference between gaze (M = 0.78, SD = 0.34) and keyboard (M = 0.003, SD = 0.013); t(29) = 12.62, p < 0.001, between gaze and mouse (M = 0.15, SD = 0.26), t(29) = 6.36, p < 0.001, and between keyboard and mouse, t(29) = -3.09, p = 0.004.



(a) Across all participants **(b) Mixed-mode interaction**

Figure 7: Analysis of usage frequency in the free-use study.

Taskwise Analysis. We analyzed the completion time and use of input device on a per task basis. Participants took the maximum time to find the bug in Task 2 (*M* = 344.23, *SD* = 198.69, 95% CI [417.82, 270.64]) and the least in Task 1 (*M* = 131.40, *SD* = 116.14, 95% CI [174.42, 88.38]). Task 3 took almost the same time as Task 1 (*M* = 134.37, *SD* = 100.49, 95% CI [171.59, 97.15]). Task 4 took the second longest time to uncover the bug (*M* = 224.30, *SD* = 130.66, 95% CI [354.96, 93.64]). There was not much difference in usage frequency between the tasks (Figure 7a). Gaze was most frequently used in Task 1 (*M* = 0.79, *SD* = 0.36) followed by mouse (*M* = 0.21, *SD* = 0.36) and then by keyboard (*M* = 0.002, *SD* = 0.01). The same trend was found for Task 2 with gaze being the most used (*M* = 0.83, *SD* = 0.28) followed by mouse (*M* = 0.17, *SD* = 0.28) and zero frequency for keyboard. Task 3 also had the highest gaze frequency (*M* = 0.87, *SD* = 0.26), followed by mouse (*M* = 0.13, *SD* = 0.26) and zero frequency of keyboard use. Finally, Task 4 also had the highest frequency use for gaze (*M* = 0.85, *SD* = 0.29), followed by mouse (*M* = 0.14, *SD* = 0.28) and then keyboard (*M* = 0.01, *SD* = 0.05).

Feature Usage Analysis. We analyzed the frequency of use of each navigation feature supported in CodeGazer (Table 1). Our results show that the most commonly used navigation was the "Go to Definition" action (*M* = 25.75, *SD* = 8.67, 95% CI [22.54, 28.96]), followed by the "Go Back" action (*M* = 14.93, *SD* = 9.81, 95% CI [11.3, 18.56]). The next two navigation actions used with almost the same frequency were the "Go to File" (*M* = 4.75, *SD* = 3.91, 95% CI [3.30, 6.20]) and "Go To Usage" (*M* = 4.36, *SD* = 3.81, 95% CI [2.95, 5.77]) actions. The two least used navigation actions were the "Go to Method" using a method list action (*M* = 0.82, *SD* = 1.44, 95% CI [0.29, 1.36]) and the "Go Forward" action (*M* = 0.39, *SD* = 0.79, 95% CI [0.10, 0.68]) .

Overall Experience. Many participants found CodeGazer easy and intuitive. Ten participants used the word "intuitive" and twelve used the word "easy" when describing their experience of using CodeGazer. Six of the 28 participants said that they took time to get used to the system. Among the learning factors, most prominent was the positions of the confirm buttons and the scrolling with gaze; in particular, participants found it hard to stop scrolling while reading code. Most said that they felt no fatigue when using CodeGazer; only three participants reported fatigue during the study.

Applicability Scenarios. Twenty two out of 28 participants said that they see CodeGazer as being most applicable in debugging scenarios. They said that they liked the ability to *quickly jump* across files and then *jump back* while following the logical structure of code. They liked not having to keep track of their current file location, as they normally would, when using the open tabbed file editors to navigate. The tabbed code browsing option was the most habitual navigation practice among participants.

CodeGazer's Effectiveness as a Navigation Tool. Most participants found navigating with CodeGazer to be quick and easy. In particular, the "Go to Definition" action with gaze was the most liked feature among the participants (RQ3), with 16 participants stating it as the feature that worked best for them. Six participants said that they liked the "Go Back" feature. Generally participants liked the idea of being able to *jump* into the definition of a method and then quickly *jump back* if they did not find what they were looking for. The following are some of the participants' responses on being asked: "What worked well with you for CodeGazer?"

P1: I think navigating, in general. It felt very natural.

P4: I think the quick commands. Such as the "Go Back" one. If I don't have my hands on the keyboard, it is really quick.

P5: Navigation was really easy. I tried the mouse one time and it feels so hard. Navigation was best with the tracker.

P6: I quite liked the idea of being able to jump into the declaration of a method, so that I could see what's inside without having to navigate through the files. I think that probably is the part of it that I liked the most.

P8: I liked the fact that I didn't have to use my hands to look through the code.

Users' willingness to use CodeGazer. Eight participants said that they would use CodeGazer for their own development work. Five said they probably would use it once they get used to it more, while another five said they would not use it. Ten participants said that they would probably not use CodeGazer for heavy development work, but would consider using it for general code exploration, especially for unfamiliar code.

Perceived Speed Compared to Mouse and Keyboard. Fifteen participants perceived gaze to be slower than the mouse, four perceived it to be faster, while five said that it was the same speed. The remaining four said they were not sure. Twenty four participants perceived gaze to be faster than the keyboard. One participant said that keyboard was faster while the remaining three simply said they found keyboard "easier."

Reasons for Choosing Gaze. Thirteen participants specifically said that they chose gaze because they wanted to "try it out," and another three said that they chose gaze "because it is fun." This shows that they initially chose gaze for novelty. A number of those participants, however, continued using mostly gaze and said that they continued to do so because it felt natural and easy. The following are quotes from the participants' responses:

P1: Initially I wanted to try using gaze. I thought it would feel forced but it didn't. It felt quite natural.

P3: It was more natural since the colors were already there on screen. It felt easier to match based on colors than to use the keyboard.

P4: I just wanted to try it. But, once I got used to it, the functionality was good, but I think the scrolling was hard.

P10: I wanted to try it out since it is quite cool. It seems to be effective too - It did not impair my ability in any way, so I thought I might stick to it anyways

P15: Initially I had my hand on the mouse, thinking that if I lose control with the gaze, I'd switch to the mouse, but then I found it very accurate and very fast. I can put my hands on my chin, because that kind of helps me think better. I think I was absorbed into using it, because it so intuitive.

P20: I feel like there is some convenience to just seeing your target and then just using your eyes to get into your target.

P26: I thought I would try gaze, but if it's really not working then I'd switch to something else. I had to do that only for scrolling at times.

Limiting Features. Participants found scrolling with gaze by far the most difficult, with 16 participants stating it to be the feature which did not work well for them in the system. Participants had problems when the scrolling would start as they looked at code at the bottom or top of the editor where the scroll bars are positioned (see Figure 1). They found scrolling with gaze to be much slower than scrolling with the mouse, and also found it hard to stop scrolling while reading code. Gaze scrolling is an optional feature of CodeGazer and people were comfortable scrolling with keyboard or mouse instead. The next most common problem was the difficulty of some participants to distinguish between the two green colored buttons at the bottom of the right margin (see Figure 1). Eleven participants stated that they got confused between these two shades of green and suggested that these two buttons should be replaced with more contrasting colors.

Discussion

Our usage frequency results show that participants used gaze most of the time during the one-hour free-use study session. As Figure 7b shows, there is no observable trend of gaze usage decreasing over time. On the contrary, the usage of gaze tended upward in the study sample from Task 1 to Task 4. Most participants expressed genuine interest in using features of CodeGazer for their own work (RQ2), to a degree that was unlikely to come just from novelty, social desirability or acquiescence bias. We are aware that the threats to validity, especially of novelty bias, cannot be fully mitigated by the nature of this study and the results should be read in that light. However, by conducting two user studies with different methodologies and a mix of different quantitative and qualitative measures, we obtained a differentiated picture of usability with consistently positive results.

Our feature usage analysis shows that the "Go to Definition" feature was most commonly used and best-received, followed by the "Go Back" feature. The results indicate that supporting users to literally "look up" new or previous information with their gaze works well (RQ3). Related work on the usage of code navigation shows that "Go to Definition" is generally the most used navigation feature in an IDE [35]. This is consistent with our results and suggests that gaze interaction works well for features that are needed most.

6 CONCLUSION AND FUTURE WORK

Source code navigation is a common activity during software development and maintenance. Software development tools such as IDEs provide options to programmers for speeding up navigation actions through context menus that are activated and accessed using either a mouse or keyboard shortcuts. These require users to position the mouse pointer or the caret at the point of interest in code before they can navigate to it. This is not as intuitive as pointing with gaze, especially during code reading and comprehension tasks where users are more focused on exploring the logical structure of the code.

Here we have presented CodeGazer, a plug-in for the IntelliJ IDE which offers navigation features for purely gaze-based code navigation: "Go to Definition," "Find All Usages" of an identifier, navigate to a usage, file or method and move back and forth between visited points in code. CodeGazer uses a gaze confirmation system [30] to disambiguate between identifiers which are close together, successfully addressing the problems of gaze tracking inaccuracy and Midas touch. An evaluation of the system indicates that it is comparable to keyboard in terms of speed (although slower than mouse), is accurate and has good usability. As future work, a cost model such as GOMS [19] could be applied to investigate theoretical performance limits. The contribution points to potentials beyond simply code navigation with gaze: code navigation is a stand-in for many activities where users will accept gaze only if it delivers clear advantages over the mouse.

REFERENCES

[1] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies* 4, 3 (2009), 114–123.

[2] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction* 24, 6 (2008), 574–594.

[3] Lonni Besançon, Paul Issartel, Mehdi Ammi, and Tobias Isenberg. 2017. Mouse, tactile, and tangible input for 3D manipulation. In *CHI Conference on Human Factors in Computing Systems*. ACM, 4727–4740.

[4] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI Conference on Human Factors in Computing Systems*. ACM, 2503–2512.

[5] Yiu-ming Cheung and Qinmu Peng. 2015. Eye gaze tracking with a web camera in a desktop environment. *IEEE Transactions on Human-Machine Systems* 45, 4 (2015), 419–430.

[6] Heiko Drewes and Albrecht Schmidt. 2007. Interacting with the computer using gaze gestures. In *IFIP Conference on Human-Computer Interaction (INTERACT)*. Springer, 475–488.

[7] Len Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT Professional* 2, 3 (2000), 17–23.

[8] Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. 2014. EyeDE: gaze-enhanced software development environments. In *CHI Extended Abstracts on Human Factors in Computing Systems*. ACM, 1555–1560.

[9] Dan Witzner Hansen, David JC MacKay, John Paulin Hansen, and Mads Nielsen. 2004. Eye tracking off the shelf. In *Symposium on Eye Tracking Research & Applications*. ACM, 58–58.

[10] Dan Witzner Hansen and Arthur EC Pece. 2005. Eye tracking in the wild. *Computer Vision and Image Understanding* 98, 1 (2005), 155–181.

[11] Dan Witzner Hansen, Henrik HT Skovsgaard, John Paulin Hansen, and Emilie Møllenbach. 2008. Noise tolerant selection by gaze-controlled pan and zoom in 3D. In *Symposium on Eye Tracking Research & Applications*. ACM, 205–212.

[12] John Paulin Hansen, Anders Sewerin Johansen, Dan Witzner Hansen, Kenji Itoh, and Satoru Mashino. 2003. Command without a click: Dwell time typing by mouse and gaze selections. In *IFIP Conference on Human-Computer Interaction (INTERACT)*. 121–128.

[13] Tobii Inc. 2018. Tobii Eye Trackers For PC Gaming - 4C Plus Alienware, Acer, MSI. https://tobiigaming.com/products/ [Online; accessed 15-February-2018].

[14] Toshiya Isomoto, Toshiyuki Ando, Buntarou Shizuki, and Shin Takahashi. 2018. Dwell time reduction technique using Fitts' law for gaze-based target acquisition. In *Symposium on Eye Tracking Research & Applications*. ACM, 26.

[15] Howell Istance, Aulikki Hyrskykari, Stephen Vickers, and Thiago Chaves. 2009. For your eyes only: Controlling 3d online games by eye-gaze. In *IFIP Conference on Human-Computer Interaction (INTERACT)*. Springer, 314–327.

[16] Rob Jacob and Sophie Stellmach. 2016. What you look at is what you get: gaze-based user interfaces. *Interactions* 23, 5 (2016), 62–65.

[17] Robert JK Jacob. 1991. The use of eye movements in human-computer interaction techniques: what you look at is what you get. *ACM Transactions on Information Systems (TOIS)* 9, 2 (1991), 152–169.

[18] JetBrains. 2017. IntelliJ IDEA the Java IDE. https://www.jetbrains.com/idea/ [Online; accessed 9-January-2017].

[19] Bonnie E John and David E Kieras. 1996. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Transactions on Computer-Human Interaction (TOCHI)* 3, 4 (1996), 320–351.

[20] Andrew J Ko, Htet Htet Aung, and Brad A Myers. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *International Conference on Software Engineering (ICSE)*. IEEE, 126–135.

[21] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[22] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987.

[23] Sai Anirudh Kondaveeti, Sandeep Vidyapu, and Samit Bhattacharya. 2016. Improved Gaze Likelihood Based Web Browsing. In *Indian Conference on Human Computer Interaction (IHCI)*. ACM, New York, NY, USA, 84–89.

[24] Per Ola Kristensson and Keith Vertanen. 2012. The potential of dwell-free eye-typing for fast assistive gaze communication. In *Symposium on Eye Tracking Research & Applications*. ACM, 241–244.

[25] Manu Kumar and Terry Winograd. 2007. GUIDe: gaze-enhanced UI design. In *CHI Extended Abstracts on Human Factors in Computing Systems*. ACM, 1977–1982.

[26] Manu Kumar, Terry Winograd, Terry Winograd, and Andreas Paepcke. 2007. Gaze-enhanced scrolling techniques. In *CHI Extended Abstracts on Human Factors in Computing Systems*. ACM, 2531–2536.

[27] Andrew Kurauchi, Wenxin Feng, Ajjen Joshi, Carlos Morimoto, and Margrit Betke. 2016. EyeSwipe: dwell-free text entry using gaze paths. In *CHI Conference on Human Factors in Computing Systems*. ACM, 1952–1956.

[28] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. 2007. Scents in programs: does information foraging theory apply to program maintenance?. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 15–22.

[29] Dongheng Li, Jason Babcock, and Derrick J Parkhurst. 2006. openEyes: a low-cost head-mounted eye-tracking solution. In *Symposium on Eye Tracking Research & Applications*. ACM, 95–100.

[30] Christof Lutteroth, Moiz Penkar, and Gerald Weber. 2015. Gaze vs. Mouse: A Fast and Accurate Gaze-Only Click Alternative. In *Symposium on User Interface Software & Technology (UIST)*. ACM, 385–394.

[31] Päivi Majaranta, Ulla-Kaija Ahola, and Oleg Špakov. 2009. Fast gaze typing with an adjustable dwell time. In *CHI Conference on Human Factors in Computing Systems*. ACM, 357–360.

[32] Päivi Majaranta and Kari-Jouko Räihä. 2002. Twenty years of eye typing: systems and design issues. In *Symposium on Eye Tracking Research & Applications*. ACM, 15–22.

[33] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In *International Conference on Program Comprehension*. IEEE, 25–35.

[34] Martez E Mott, Shane Williams, Jacob O Wobbrock, and Meredith Ringel Morris. 2017. Improving dwell-based gaze typing with dynamic, cascading dwell times. In *CHI Conference on Human Factors in Computing Systems*. ACM, 2558–2570.

[35] Gail C Murphy, Mik Kersten, and Leah Findlater. 2006. How are Java software developers using the Elipse IDE? *IEEE Software* 23, 4 (2006), 76–83.

[36] Brad A Myers, Andrew J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.

[37] Aanand Nayyar, Utkarsh Dwivedi, Karan Ahuja, Nitendra Rajput, Seema Nagar, and Kuntal Dey. 2017. OptiDwell: intelligent adjustment of dwell click time. In *International Conference on Intelligent User Interfaces (IUI)*. ACM, 193–204.

[38] Jakob Nielsen. 1994. *Usability engineering.* Elsevier.

[39] Abdul Penkar, Christof Lutteroth, and Gerald Weber. 2013. Eyes only: navigating hypertext with gaze. In *IFIP Conference on Human-Computer Interaction (INTERACT).* Springer, 153–169.

[40] Abdul Moiz Penkar, Christof Lutteroth, and Gerald Weber. 2012. Designing for the eye: design parameters for dwell in gaze interaction. In *Australian Computer-Human Interaction Conference (OzCHI).* ACM, 479–488.

[41] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. 2012. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. In *CHI Conference on Human Factors in Computing Systems.* ACM, 1471–1480.

[42] David J Piorkowski, Scott D Fleming, Irwin Kwan, Margaret M Burnett, Christopher Scaffidi, Rachel KE Bellamy, and Joshua Jordahl. 2013. The whats and hows of programmers' foraging diets. In *CHI Conference on Human Factors in Computing Systems.* ACM, 3063–3072.

[43] Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. 2016. Eye-Nav: gaze-based code navigation. In *Nordic Conference on Human-Computer Interaction.* ACM, 89.

[44] Javier San Agustin, Henrik Skovsgaard, Emilie Mollenbach, Maria Barret, Martin Tall, Dan Witzner Hansen, and John Paulin Hansen. 2010. Evaluation of a low-cost open-source gaze tracker. In *Symposium on Eye-Tracking Research & Applications.* ACM, 77–80.

[45] Simon Schenk, Marc Dreiser, Gerhard Rigoll, and Michael Dorr. 2017. GazeEverywhere: enabling gaze-only user interaction on an unmodified desktop PC in everyday scenarios. In *CHI Conference on Human Factors in Computing Systems.* ACM, 3034–3044.

[46] Laura Sesma, Arantxa Villanueva, and Rafael Cabeza. 2012. Evaluation of pupil center-eye corner vector for gaze estimation using a web cam. In *Symposium on Eye Tracking Research & Applications.* ACM, 217–220.

[47] Claude M Steele and Joshua Aronson. 1995. Stereotype threat and the intellectual test performance of African Americans. *Journal of Personality and Social Psychology* 69, 5 (1995), 797.

[48] Ken Neth Yeoh, Christof Lutteroth, and Gerald Weber. 2015. Eyes and Keys: an evaluation of click alternatives combining gaze and keyboard. In *IFIP Conference on Human-Computer Interaction (INTERACT).* Springer.