Javardeye: Gaze Input for Cursor Control in a Structured Editor

André L. Santos
Instituto Universitário de Lisboa (ISCTE–IUL), ISTAR-IUL
Lisboa, Portugal
andre.santos@iscte-iul.pt

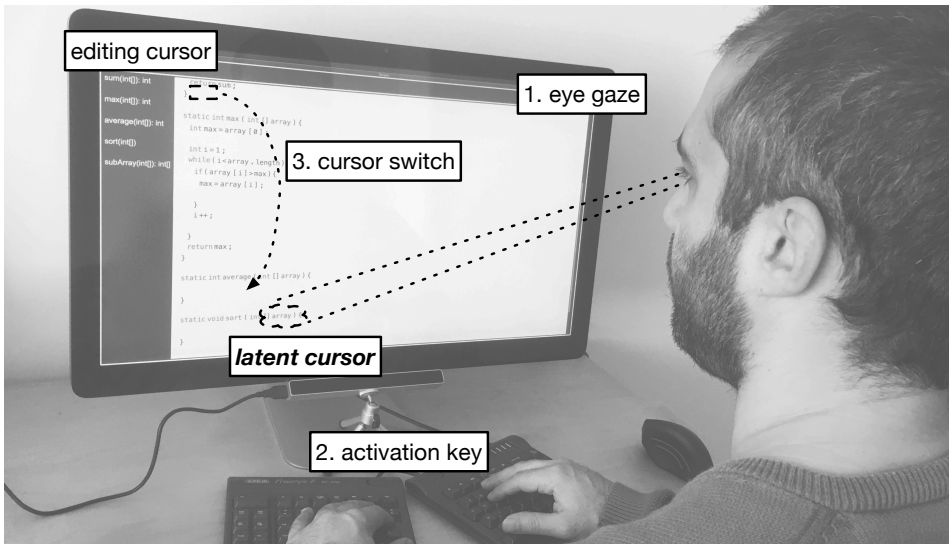

Figure 1: Latent Cursor: a secondary gaze-driven cursor locates the next editing point to be activated with a key.

ABSTRACT

Programmers spend a considerable time jumping through editing positions in the source code, often requiring the use of the mouse and/or arrow keys to position the cursor at the desired editing position. We developed Javardeye, a prototype code editor for Java integrated with eye tracking technology for controlling the editing cursor. Our implementation is based on a structured editor, leveraging on its particular characteristics, and augmenting it with a secondary—latent cursor—controlled by eye gaze. This paper describes the main design decisions and tradeoffs of our approach.

CCS CONCEPTS

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

KEYWORDS

structured editors, gaze input, navigation

ACM Reference Format:
André L. Santos. 2021. Javardeye: Gaze Input for Cursor Control in a Structured Editor. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (<Programming> '21 Companion), March 22–26, 2021, Virtual, UK*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3464432.3464435

<Programming> '21 Companion, March 22–26, 2021, Virtual, UK

ACM ISBN 978-1-4503-8986-0/21/03…$15.00
https://doi.org/10.1145/3464432.3464435

1 INTRODUCTION

Developers have to deal with large source code files and editing activity easily becomes tiresome. A considerable part of the time is spent on IDE interaction [5, 8], namely on navigating, mouse drifting, code inspection. When inspecting a part of a source code file for modification, one has to position the cursor at the desired location. This action will conventionally either involve the keyboard or the mouse.

Anecdotical evidence suggests that experienced programmers prefer keyboard in favor of resorting to the mouse, as the latter requires more movement, and hence, tends to be slower for many actions. Programmers are among the users that most intensely use the keyboard. Anecdotal evidence from a considerable number of posts in blogs and discussion forums, suggests that many programmers incur in repetitive strain injuries (RSI), affecting wrists, hands, or shoulders. The author of this paper himself has had wrist RSI.

Arrow keys in most keyboards are usually not in the most ergonomic location (at bottom right, not favoring wrist health given the necessary hand bending to reach them). On the other hand, intense mouse usage also causes problems, not only in the hands and wrists, but also in arms and shoulders. Our research aims at reducing time and physical effort for cursor positioning, when compared to the equivalent actions using keyboard and mouse (in isolation or in combination).

This paper presents a prototype code editor where the cursor location is controlled through eye gaze input, using an eye-tracking device. Our design relies on the concept of *latent cursor*, an additional "shadow cursor" in the editor for the gaze location (see Figure 1), to which the user may jump on request by pressing a key. This a gaze-supported selection technique, following the principle that *gaze suggests* and *touch* confirms. [12] We explain the main challenges and limitations for achieving this goal, and we describe the design of our prototype. We developed the prototype on top of Javardise [10], a structured editor for Java initially developed for didactic purposes. Having an underlying structured editor is an advantage to cursor positioning, given that code elements are well-defined widgets (as opposed to a stream of characters), a characteristic that facilitates locating elements.

Our main goal is to develop a technique for quickly jumping from one part of the code to another, within the same file. With the available technology, users most likely use arrow keys for shorter distances, whereas for longer ones the chances of opting for the mouse pointer are higher. Nonetheless, there are many editors with a variety of shortcuts, as well as many types of users, with different expertise, culture, etc. We propose to control cursor positioning with eye gaze, allowing a user to jump to an editing location by looking at it. This is a natural form of interaction, as one implicitly looks at a destination before pointing there, indicating a hypothetical intention of moving there.

Quick jumps may result in faster code editing. However, even if the proposed mechanism does not result in faster code editing, allowing fewer keystrokes or mouse manipulation should in principle contribute to a less tiresome editing activity. Hence, we consider that apart from coding speed, reducing physical effort is also a goal worth pursuing.

2 CHALLENGES

The design of a solution for our goals has a few challenges regarding the gaze control modality that may hinder its effectiveness.

2.1 Midas Touch Problem

One could use eye gaze to control the mouse. However, one pitfall with this approach is the Midas Touch problem [4], which consists in accidental touching activity, for instance when using gaze control for performing mouse actions (clicks or moves). This may negatively affect the user experience and invalidate designs based on eye gaze. We chose not to use the eye tracking interaction modality to replace the mouse in any way, but rather to gather complementary information to eventually trigger interaction. Jacob et al. [3] advocate for this strategy, as humans often perform saccades for quick glances, and that would lead to involuntary mouse activity.

2.2 Precision of Eye Tracking Device

Code involves fine-grained editing, and often small fonts are used in editors. Source code tokens may be as small as a single character. This raises a concern regarding the precision of eye tracking devices. Low accuracy of eye gaze data, forcing a user to wait more than a fraction of a second, or that requires tricking the natural way of looking undermines the whole experience. The precision of eye gaze depends on the eye tracking hardware being used, as well as on the quality of its calibration.

A user study with a diverse demographics of non-programmers, revealed that accuracy and precision may vary substantially among subjects—up to six-fold [1]—using two of the best eye tracking devices available (Tobii EyeX[1], SMI REDn[2]). The study concluded that for reaching reliable interaction for 75% of the users, UI designs should consider targets of at least 1.9cm width and 2.35cm height (accuracy tends to be lower in the y axis). However, more accurate users (25%) may interact well with targets as small as 0.58×0.8 cm.

The font size and face that programmers use varies according to taste and user physiology. We are not aware of a systematic survey on the matter, but from anecdotal evidence, we find reasonable to assume that the norm is to use a mono-spaced font face with size ranging between 11 and 14 points. Whereas 0.58×0.8 cm seem a reasonable target dimension for such font sizes, eye tracking systems with low precision may require a larger than usual font size for satisfactory accuracy. Apart from the size of the actual font, line spacing might have to be tuned, especially because the y-axis accuracy tends to be lower. [1]

3 RELATED WORK

EyeNav [9] is a gaze-enabled IDE that mostly resembles our approach, in the sense that eye gaze is used to navigate to the editing location. We also adopt a "look-and-click" [12] interaction technique to move the cursor. EyeNav supports selection of a range of characters, an aspect which does not straightly apply in our case because we are working with a structured editor. We plan to leverage on gaze input for selecting a set of elements, but we did not yet come up with a technique to achieve that goal, in part due to the limitations and prototypical state of Javardise.

EyeDE [2] is also a gaze-enabled IDE, but its focus is on navigation features, such as jumping to declarations, documentation lookup, and switching between files. Their tool uses gaze input to select an identifier, and in turn, activate the possible actions for that same identifier through a gaze-enabled pop-up menu. Most of the features proposed by EyeDE could be integrated with what we propose, given that their starting point is the gaze-driven selection of a program token. The main difference of our contribution to EyeNav and EyeDE is that we exploit the specific characteristics of a structured editor, which target conventional code editors holding streams of characters. We expect to achieve more gaze accuracy than these systems because we work with larger gaze target regions, as explained ahead.

CodeGazer [11] follows a similar strategy to EyeDE, but relies on a different technique for gaze-only selection called Actigaze

[1] www.tobiipro.com
[2] www.imotions.com/hardware/smi-redn-scientific

[7]. The technique relies on displaying side and upper/lower bars with large target regions, each with a different color, which maps to program elements (or menu options) that also become colored when they are within gaze range. The elements are selected by gazing the side target areas that match the colored elements. This technique is more invasive than EyeNav, EyeDE, and ours, given that it requires additional bars on the sides, and further interferes with the code layout given the frequent color changes. The main advantage of CodeGaze is that is gaze-only, rather than look-and-click.

Thomschke et al. [13] proposed to use gaze input for inter-environment switching, rather than within a same programming environment. The motivation for this approach are the mismatches between the environment actually on focus and the one we aims at interacting with, possibly leading to keyboard typing at incorrect and inconvenient places. The approach is based on matching keyboard focus with the gaze input, assuming that a developer will look to the desired region for typing. Our approach is solely addressing intra-environment gaze-based control, but in principle it could integrate well with such an inter-environment gaze control given that gaze has no effect outside our editor window.

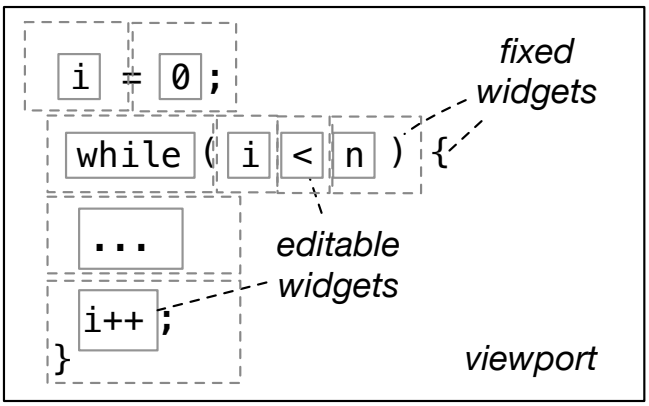

Figure 2: Javardise holds program elements in either editable or fixed widgets. The dashed regions represent the augmented bounds of widgets that will be gaze targets for selecting them.

4 DESIGN

We address the previously described design challenges exploiting the particular characteristics of the implementation of our structured editor.

4.1 Anatomy of the Structured Editor

Our prototype was built on top of a structured editor that resulted from our previous work on Javardise. [10] The implementation of the structured editor is based on a containment hierarchy of widgets that hold elements of the program (tokens representing expressions or language keywords). The editing activity consists of typing on the text widgets, even if that is not fully realized by the user.

As expected, the user sets the focus on these editable text widgets using either mouse or keyboard. As with normal UI toolkits, the widgets have well-defined boundaries that can be easily mapped to display coordinates (see Figure 2). We rely on this aspect in order to have more coarse-grained elements as eye gaze targets, in contrast to single-character cursor positioning. This alleviates the accuracy problem, given that gaze will target whole code tokens that have larger bounds.

Another characteristic of Javardise is that editable program tokens (e.g., identifiers, operators) and fixed program tokens (e.g., brackets, semi-colons) are handled differently and may be unambiguously discerned (see Figure 2). If we exclude the fixed widgets, we have fewer elements as possible gaze targets, and the target region of the editable widgets may be expanded to overlap the fixed widgets. Having larger target regions also mitigates the accuracy problem.

4.2 Latent Cursor

Our proposed solution comprises a secondary "cursor", which we refer to as the *latent cursor* (recall Figure 1), in addition to the regular *editing cursor*. The latent cursor is not actually a cursor where one may start entering text straight away, but a location that may immediately become the editing cursor. The latent cursor moves according to the user's eye gaze, and it comes into play when a specific key (or combination) is pressed. This activates switching the editing cursor to the latent cursor, whereas the new position of the latent cursor location will go on being driven by gaze.

We maintain the regular editing cursor working normally, whereas the gaze cursor has a very lightweight distinctive appearance, in order to be the least distractive as possible, but nevertheless, visible (see Figure 3). We chose to have a slightly distinct text background for marking the gaze location. However, we did not actually investigate which would be the best options in HCI terms. Nevertheless, this aspect is suitable to be easily changed.

The position of the latent cursor may be affected by eye tracking accuracy, and hence, it could possibly be located not at the exact desired spot. However, as with the case that we might miss an exact character position target with the mouse and further adjust with keys, one can also trigger cursor switch to the latent cursor to an approximate desired location and further adjust with arrow keys. This is not the ideal situation, but it may still involve less activity than manual positioning (2-3 keystrokes vs. switching to the mouse).

4.3 Beyond the Viewport

Code files are often large, and a programmer will need to move around within a file. We considered two eye gaze-based mechanisms for changing what is visible in the viewport.

In order to perform longer movements within the same file, we provide the outline section. The latent cursor also moves to this area, and navigation is triggered in the same way as explained. By pressing the activation key, the viewport jumps to the selected method, as in a regular IDE outline view.

In order to reach adjacent regions to the viewport, Javardeye supports automatic scrolling when gazing and dwelling over the upper and lower areas of the editing area (see Figure 3). At the edge of the editing area the scroll moves at a single step at a time,

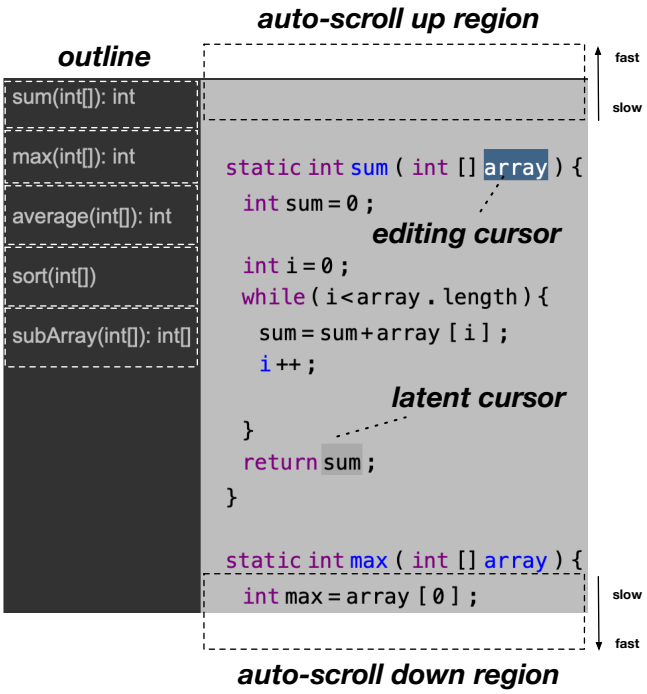

Figure 3: User interface of Javardeye. Dashed parts are target regions for gaze-driven activity: method focus on the outline section on the left; up/down scrolling on the top/bottom edges of the viewport.

whereas as the distance from the edge towards outside increases, the scrolling speed also increases.

In contrast to cursor and outline jumps, scrolling does not use an activation key, and hence, the Midas touch problem may emerge here. However, automatic scrolling occurs without loosing the current editing cursor position (an adaptation of the discrete scrolling with gaze-repositioning technique [6]). Given that quick glances will result only in small scrolling steps without loosing the cursor, we speculate that it should not significantly harm the editing experience.

4.4 Implementation

We used an EyeTribe[3] device (now discontinued) as the eye-tracking hardware to implement the prototype. EyeTribe is launched as a local server, to which other processes may establish connections to receive gaze data. They provide SDKs for C, C++, and Java for communicating with the server. Given that Javardise is built in Java, the integration was smooth in this respect. We allow the latent cursor to be turned on and off at any point in time. Regarding configuration, we allow to define the key for triggering the cursor switch and dwelling time for scrolling.

Using the available EyeTribe device of our lab, we did not manage to have excellent calibration results. This resulted in moderate gaze accuracy, implying that the prototype was usable only with a large font (> 24 points). In principle, if using better hardware, such as the devices manufactured by Tobii, we can manage to operate with font sizes that are closer to normal standards. We used EyeTribe for this first prototype merely due to practical reasons. Among the available hardware at our research center, EyeTribe has the easiest to integrate with Mac OS, since the available Tobii hardware did not have out-of-the-box drivers for it.

[3] theeyetribe.com

5 CONCLUSIONS

The development of Javardeye led us to conclude that the combination of eye tracking and structured code editors has potential as an effective multi-modal interaction technique. The main advantages relate to having code elements as widgets, with well-defined boundaries that may stretch over to non editable tokens to allow a larger gaze target.

So far, we concentrated mostly on cursor control, but additional gaze-based features could be developed. We did not implement any technique for inter-file navigation, but we foresee that the techniques from other works, such as navigate to definition (e.g., [2]), could be seamlessly integrated with our technique. We are working on a technique for using the latent cursor to form a selection in combination with the active cursor.

Once the ability of performing selection (for copy/cut-paste) is achieved, we should be close to a thorough gaze-based interaction with the code editor. At that point, we plan to carry out a user study to evaluate the usability of Javardeye.

ACKNOWLEDGEMENTS

This work was partially funded by *Fundação para a Ciência e Tecnologia* (FCT / Portugal) by the project UIDB/04466/2020. We thank Instituto Universitário de Lisboa and ISTAR-IUL for their support, as well as the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Anna Maria Feit, Shane Williams, Arturo Toledo, Ann Paradiso, Harish Kulkarni, Shaun Kane, and Meredith Ringel Morris. 2017. *Toward Everyday Gaze Input: Accuracy and Precision of Eye Tracking and Implications for Design.* Association for Computing Machinery, New York, NY, USA, 1118–1130. https://doi.org/10.1145/3025453.3025599

[2] Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. 2014. EyeDE: Gaze-Enhanced Software Development Environments. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems (CHI EA '14).* Association for Computing Machinery, New York, NY, USA, 1555–1560. https://doi.org/10.1145/2559206.2581217

[3] Rob Jacob and Sophie Stellmach. 2016. What You Look at is What You Get: Gaze-Based User Interfaces. *Interactions* 23, 5 (Aug. 2016), 62–65. https://doi.org/10.1145/2978577

[4] Robert J. K. Jacob. 1990. What You Look at is What You Get: Eye Movement-Based Interaction Techniques. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90).* Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/97243.97246

[5] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. 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. https://doi.org/10.1109/TSE.2006.116

[6] Manu Kumar and Terry Winograd. 2007. Gaze-enhanced scrolling techniques. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, Newport, Rhode Island, USA, October 7-10, 2007*, Chia Shen, Robert J. K. Jacob, and Ravin Balakrishnan (Eds.). ACM, 213–216. https://doi.org/10.1145/1294211.1294249

[7] Christof Lutteroth, Moiz Penkar, and Gerald Weber. 2015. Gaze vs. Mouse: A Fast and Accurate Gaze-Only Click Alternative. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15).*

Association for Computing Machinery, New York, NY, USA, 385–394. https://doi.org/10.1145/2807442.2807461

[8] Roberto Minelli, Andrea Mocci and, and Michele Lanza. 2015. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, 25–35.

[9] Stevche Radevski, Hideaki Hata, and Ken-ichi Matsumoto. 2016. EyeNav: Gaze-Based Code Navigation. In *Proceedings of the 9th Nordic Conference on Human-Computer Interaction, Gothenburg, Sweden, October 23 - 27, 2016*. ACM, 89. https://doi.org/10.1145/2971485.2996724

[10] André L. Santos. 2020. Javardise: A Structured Code Editor for Programming Pedagogy in Java. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (<programming> '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

[11] Asma Shakil, Christof Lutteroth, and Gerald Weber. 2019. CodeGazer: Making Code Navigation Easy and Natural With Gaze Input. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300306

[12] Sophie Stellmach and Raimund Dachselt. 2012. Look & Touch: Gaze-Supported Target Acquisition. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. Association for Computing Machinery, New York, NY, USA, 2981–2990. https://doi.org/10.1145/2207676.2208709

[13] Astrid Thomschke, Daniel Stolpe, Marcel Taeumel, and Robert Hirschfeld. 2016. Towards Gaze Control in Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop (PX/16)*. Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/2984380.2984384