Towards Gaze Control in Programming Environments

Astrid Thomschke[⋆] Daniel Stolpe[⋆] Marcel Taeumel [⋆ †] Robert Hirschfeld [⋆ †]
[⋆]Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany
[†]Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
{astrid.thomschke, daniel.stolpe}@student.hpi.uni-potsdam.de
{marcel.taeumel, robert.hirschfeld}@hpi.uni-potsdam.de

ABSTRACT

Elaborated gaze tracking devices are hitting the consumer market. This gives an existing human-computer interaction technique the chance to be widely applied in software applications. Programmers can benefit from this development. They tend to work on multiple or large screens to interact with diverse tools in parallel. When programmers switch between reading and typing, the keyboard focus might not be where expected. Such distractions leave the programmer dissatisfied. Gaze information can help to determine which tool a programmer focusses on.

Our goal is to explore the use of gaze information for programming environments. Specifically, we investigate a case where a programmer's view focus and the intended keyboard focus correlate. For specific programming tasks, our work shows that it is beneficial to set the keyboard focus to a programmer's view focus.

CCS Concepts

•**Software and its engineering** → **Object oriented development;** *Software prototyping;* Object oriented frameworks; Integrated and visual development environments;

Keywords

Gaze Tracking, Gaze Control, Programming Environment, Integrated Development Environment

1. INTRODUCTION

In a programming environment, users perform various tasks in multiple software environments. Situations can occur where the keyboard focus is not set to the expected input field. These cases create security flaws, distractions and dissatisfaction. A password can end up in the development chat instead of the password field. A shortcut can close the debugging session instead of a browser tab. Sometimes it is even unclear if text was typed anywhere at all.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PX/16, July 18 2016, Rome, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4776-1/16/07. . . $15.00

DOI: http://dx.doi.org/10.1145/2984380.2984384

A programmer's software environment necessitates a huge amount of information to be visible right next to each other. Figure 1 visualizes a multi-screen setup consisting of diverse tools. The programmer has to combine a complex set of information in order to write code. This includes reading code and documentation, gathering runtime information, and keeping the specification in mind. In addition, programming is a collaborative task that adds chat windows, video calls, and an email inbox to the environment.

Programmers have to interact with multiple tools and diverse content in parallel. This requires careful organization of available screen space. As a result, it allows programmers to display all the available information side by side. Software development is a profession that makes extensive use of keyboard shortcuts to navigate such a complex environment. This often leads to avoiding pointing devices where possible. As a consequence, visualizing keyboard focus becomes even more important.

This problem can easily be solved for situations that require modal interaction, such as password requests and points of single access. For example, all content apart from the modal dialog can be obstructed to guide the users view. However, this is not a good strategy for workflows that require more than one source of information to be visible. A second way is to clearly highlight the focused object. Visualizations such as shadows, borders, or blinking cursors can quickly become a design obstacle.

For this work, we assume that the keyboard focus is closely connected to our view focus. More specific, we assume that it is beneficial to derive the keyboard focus from a user's gaze. We draw our assumption from observations made in human-to-human communication. Pointing in the direction we want our opponent to look is not always necessary. For example, a person can deduce that another person describes a direction from the conversational context. Explicit pointing is rarely required when context information is available. Following a person's gaze already provides sufficient information about the center of the conversation.

We imagine a programming environment capable of combining context information with gaze to derive the correct

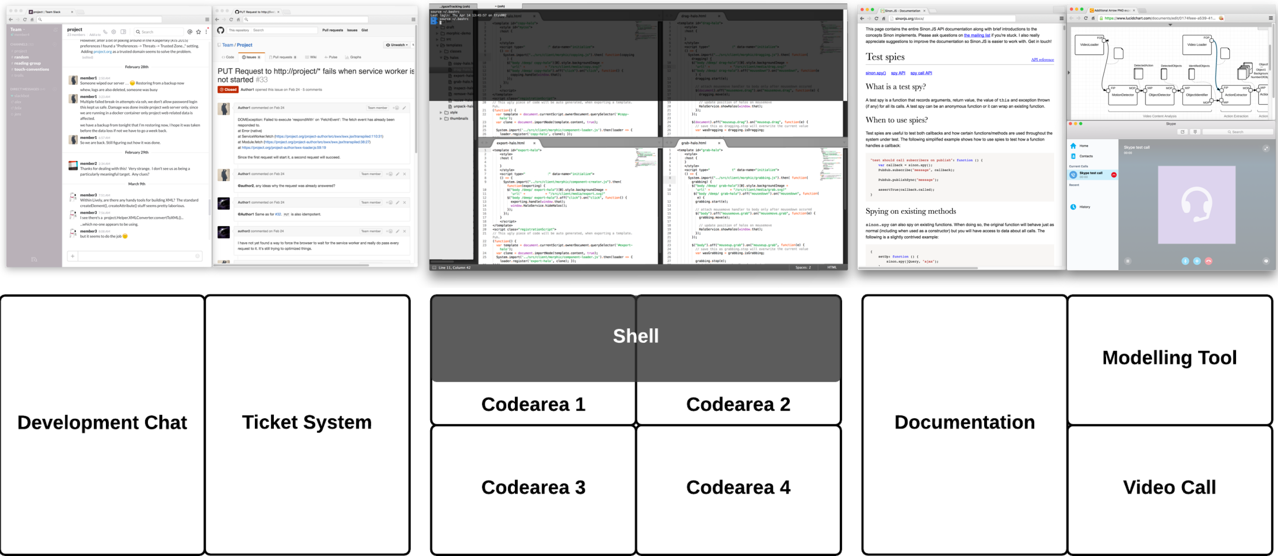

Figure 1: Example of a programmer's screen setup at the top has been simplified in the bottom view. It includes the code and a shell on the central screen, contextual information adjacent to it and collaboration tools at the very left and bottom right screen.

keyboard focus. The goal of this project is to consider a programmer's gaze when setting the keyboard focus. We aim to reduce focusing errors that occur when a programmer switches between reading information and creating code.

2. BACKGROUND

For our research, we distinguish between a general and a working hypothesis. In this paper, we only evaluate the latter.

Our general hypothesis is that it is possible to derive the intended keyboard focus from a users gaze. Our working hypothesis here is that it is possible to set the keyboard focus to exactly the input field the user is looking at. This excludes cases in which gaze focus and intended keyboard focus differ.

We limit our hypotheses to programming environments.

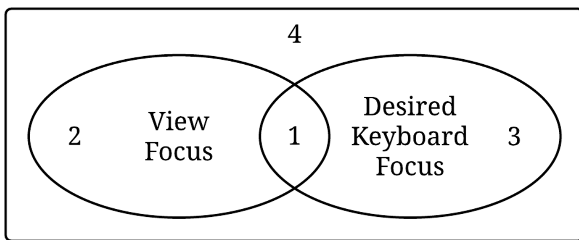

Figure 2: View focus and desired keyboard focus corellate, but only in one of four cases.

Figure 2 explains the difference between our general and working hypothesis. It shows that the view focus and the desired keyboard focus only correlate in one specific case. When deciding at any point if the keyboard focus should be set to an object in a certain area, four situations are possible:

1. The object is in the viewed area and is the desired keyboard focus.

2. The object is in the viewed area but is not the desired keyboard focus.

3. The object is not in the viewed area but is the desired keyboard focus.

4. The object is not in the viewed area and is not the desired keyboard focus.

Despite describing similar states, the four points underline different use cases. First, in order to derive keyboard focus from view focus, an environment has to be able to set the keyboard focus where the users view focus is. Second, it must be possible to detect when the keyboard focus should not be set if the programmer is looking away from its desired target. This happens e.g. when using a template to create content. Third, it must be able to set a keyboard focus to a target the user is not looking at. When browsing a list of objects with content, a person will constantly look at the target where the content is displayed. The list itself, however, has to catch the keyboard focus. Fourth, the environment must prevent cases where the keyboard focus is set to a target that is not viewed.

Our work focuses on the first case. We assume that programmers can deal with an environment that sets the keyboard focus to what they look at. Within the scope of our work lies evaluating whether this is a reasonable focusing strategy or if it distracts the user. We also aim at evaluating different focusing strategies for the purpose of creating textual. Those are introduced in the next section.

3. RELATED WORK

Our work benefits from research in different areas. On a technical level, gaze-tracking accuracy constantly improves. On a medium level, a set of gaze input techniques has been evolved and tested for a number of tasks. Finally, gaze-enhancement for specific use cases in common software and programming environments has been evaluated.

Current gaze tracking systems based on infrared light achieve accuracies of 0.4°, or an average distance of 24mm from the actual target with a 60cm distance to the screen [2]. RGB image based solutions achieve accuracies of 2°[11]. This means we can assume that it is possible to focus a normal text area in an integrated development environment (IDE), but will not succeed in reliably focusing buttons, or even password input fields, without further algorithmic improvements. Vertegaal also stated the problem of accuracy, but was able to confirm that Fitts law applies to gaze tracking techniques [10]. We can therefore assume the scalability of gaze control to large and multiple screens.

Leveraging gaze as a control technique, Zhai notes that "the eye has not evolved to be a control organ" [12]. However, he sees pointing as a acceptable tasks, eventually finding warping to be a preferred pointing technique. Warping implements a focus point that gravitates towards near objects. He distinguishes liberal warping that is constantly changing the focus point, and conservative warping that only changes focus using a trigger. He finds that liberal warping was preferred by his users.

When transferring the task of pointing to more complex use cases, conservative approaches are preferred. In Eye-Point, Kumar concentrates on selection tasks. He finds that his users preferred the look-press-look-release, which allows selecting an element only while a trigger is being pressed [6]. Kumar also suggests scrolling as a scenario where gaze control can be useful [7]. Penkar finds benefits from navigating hypertext with gaze [8]. Penkar and Kumar extend a list compiled by Jacob that also includes continuously displaying attributes, moving objects and invoking menu commands. Jacob also introduces listener windows that are capable of catching keyboard input once a user looks at them [5]. We assume that listener windows are the most prevalent scenario in a programming environment.

In the areas of programming environment, Glücker describes EyeDE as a gaze-enhanced software development environment. He shows that invoking menu commands with gaze is useful for jumping to method declarations and back, looking up documentation and expanding function bodies. Glücker also suggests fine-granulated pointing tasks such as text highlighting. He also implements a gaze-based file switch, but concentrates on cases where files are stacked instead of arranged side-by-side [3].

4. IMPLEMENTATION

We suggest implementing a gaze-enhanced programming environment on four levels, as visualized in Figure 3. Each level allows for context specific adaptions.

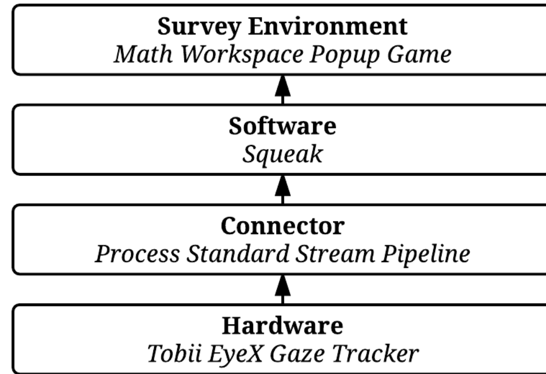

Figure 3: We implemented our gaze-enhanced environment on four layers.

4.1 Hardware

On the hardware level, current gaze tracking systems make use of eye and head tracking techniques [1]. Eye tracking calculates the pupil position relative to the eye. Head and gaze tracking compute a position relative to a device and typically require calibration. We use a gaze tracking system for this project. For future projects, adding pure head tracking to this level is considered in Section 7.

There are two main techniques to measure the pupil position required for gaze. First, a standard image can be examined for features that reconcile eyes. The more accurate method uses infrared light to discover pupils [11]. However it might not work with certain illuminations. We decided for an infrared-based approach, as programmers typically work in well-lit environments.

To measure the head position, three approaches can be taken. First, it is possible to fixate a face in relation to the screen, making head position tracking obsolete. Second, a head-mounted device can be used to measure tilt, rotation and distance. Third, head position can be recovered based on image processing. However, when using two cameras, it is possible to bypass head position calculation, as in such cases the pupil position can sufficiently be utilised to calculate its relation to a device. We chose the last approach, as it combines accuracy with freedom of movement. We used a Tobii EyeX Controller[1] to gather gaze information.

[1] http://www.tobii.com/xperience/products/

4.2 Connector

The information gathered from the hardware can be preprocessed in order to communicate with the programming environment. It is possible to establish a connection via low level adapters that base upon inter-process communication. A lighter level can be used to communicate gaze information to a remote system, e.g. an application running in the browser. We implemented an inter-process communication using process standard stream pipes.

On this level, the gaze information can be brought into a processable format for the programming environment. It is also possible to modify the frequency of position updates in order to improve performance. If required, this level of abstraction allows manipulating the gaze coordinates.

4.3 Software

We chose to implement our chosen aspect gaze control in the Squeak [4] programming environment. The software level allows us to implement our working hypothesis, by setting the keyboard focus to the currently viewed object.

It is required to transpose the gaze coordinates received from the connector to the screen bounds of the programming environment. If the environment is taking the full screen space, this step can be avoided.

It is also possible to implement advanced pointing techniques. We chose to work in a live programming environment, as it can easily be adapted to test different techniques and scenarios. It also allows simulating a combination of all tools a programmer requires within one environment.

Being a live programming environment, Squeak provides us with information about the bounds of all relevant objects on the screen. It also allows testing different pointing techniques while working in the same environment. We experimented with different warping approaches, as well as cursor and focus visualizations.[2]

4.4 Survey environment

We implemented a game that allows us to test the hypothesis we implemented on the software level. Our environment is supposed to test cases where a programmer has to rapidly switch between view targets. This simulates gathering information from multiple places, as it is typical in a programming environment. A programmer is also working under a certain cognitive load, solving complex tasks. Since a programmer mainly works with his keyboard, the test task should mainly consist of typing.

Combining those properties, we implemented a game where, in random screen positions, workspaces appear. To trigger the next workspace, a math question has to be solved. This creates a certain cognitive load for the participant. The answer has to be typed into the workspace, after it is focused. Figure 4 shows an example workspace.

The benefit of having this game in a live programming environment is that we can easily adapt it in order to test scenarios two, three and four of Figure 2.

[2]See https://www.youtube.com/watch?v=SNm0B0PTYQI for a demonstration of our prototypical implementation.

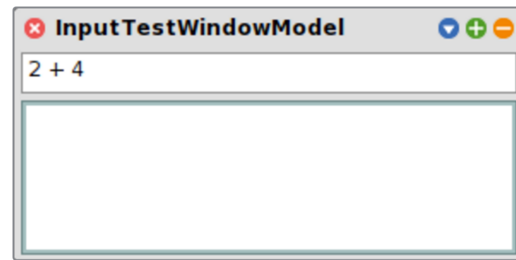

Figure 4: The user has to type the answer to the question in the top line into the workspace body.

5. EVALUATION

We asked twelve programmers to play the implemented game. Each participant took two runs, one using the mouse as pointing device, and one where keyboard focus was continuously set to gaze focus. A run defined as 40 seconds, where numbers between zero and ten were added. We allowed for five questions of adjusting to the environment. Before running with gaze control, participants calibrated the gaze tracking system. For each run, we measured the number of correct and wrong answers. The accumulated amount of solved questions allowed us to see if pointing with gaze significantly differed from using a mouse. The amount of correct answers gave us information about the cognitive load. A significant increase of wrong answers could point us to increased cognitive load, or distraction. We conducted retrospective interviews in order to learn about participants feelings towards gaze control and also collect further ideas.

Users averaged at 27 answers per minute using a mouse, and 25.5 using gaze. They always gave one or less wrong anwers. There was neither a significant drop in the number of solved questions nor increase in wrong answers. Thus we can assume that participants were able to focus with gaze without performance drops. During the interview, we found that most participants were impressed by the accuracy of gaze control. The quickest person stated being on the verge of feeling “cheated at” whenever the measured gaze coordinates were slightly off. One participant continued to work in our environment for several minutes. He found that he felt “exhausted after working with the gaze tracker for a longer time”. Three participants stated that they had to focus the workspace center instead of the question itself, which was placed near the top border of the workspace. Due to inaccuracies, looking at an objects edges would not set the focus the the object.

All participants gave us further ideas for gaze interaction. Everyone had a blink-to-confirm interaction in mind, but quickly withdrew the idea when simulating it, finding it too similar to the natural reflex of blinking. Users also suggested gestures such as nodding or shaking the head for confirmation and denial. One suggested allowing for concentrated work in a single workspace by having a parking area for a gaze cursor. When looking into the top right corner, for example, the gaze cursor could be parked or picked up.

6. DISCUSSION

We benefitted from working in the same programming environment that we conducted our study in. Our user study and the gaze control techniques were both implemented in Squeak. We implemented our evaluation game using the described focusing strategies ourselves. We later combined the experiences we made here with the retrospective interview results.

6.1 Refinements

During our work we kept adapting our system to test non-ideal settings and tweak the programming experience.

100ms threshold.

Zhai arguments against a time threshold that controls when a target should be selected and instead suggests liberal warping [12]. We found it helpful though to account for rapid eye movements and errors to set the threshold to less than 100ms until an object is focused.

No pointer.

We felt most comfortable with not moving a cursor at all for tasks similar to programming. Instead, we only set the keyboard focus to the currently viewed input field. When working with text, fine granular movements of the mouse cursor as described by Zhai [12] are not required as programmers rather use the keyboard here.

Centered information.

We found that it was simpler to focus elements when information was placed in the center rather than at the edges of a text area. The gaze tracking coordinates differ about 2cm from the actual gaze. When focusing adjacent workspaces, this inaccuracy makes it difficult to determine which workspace a programmer is looking at. When information is placed in 2cm distance to an area's border, keyboard focus can be set with more confidence.

Target size.

Our setup was too inaccurate to focus a single input line. This affects single-line inputs such as passwords or chat clients. However, adding a half line margin makes it possible to automatically focus those inputs.

6.2 Opinions and feelings

There were some major pain points with gaze-based focusing, whose origins are open for discussion.

Importance of gaze accuracy.

The hardware setup we used seemed sufficiently accurate. We let some of our participants skip the calibration to simulate a long coding session that includes leaving the workspace. This created just slight offsets, which had immense effect. Users felt cheated at and annoyed. They wanted to adapt to the offset and demanded to see which caze coordinates the environment had computed.

It seems that usually users can adjust to a slight offset of pointing devices. For gaze control, however, we see that accurate gaze tracking is immensely important.

Cognitive load.

Working in a gaze-enhanced environment felt more straining for the eyes than usual.

We assume that, when eyes are to a certain extend used as a control organ, cognitive load is increased. The reason might be that users become aware of the usually subsconcious pupil movements.

6.3 Gaze gestures

We considered the use of head and gaze induced gestures our participants suggested during evaluation in Section 5.

Parking lot.

Some programming tasks consist of long reading passages where a keyboard focus is not required at all. In order to decrease cognitive load as reported in 6.2, a sensitive area could allow for pausing gaze control. When viewed, such an area could toggle automatic keyboard focus setting.

Such a technique introduces a modal interaction. This requires cautious implementation in order to reduce the danger of confusion by obscured state.

Head gestures.

In order to confirm or deny actions, simple gestures like nodding or shaking the head were suggested. Such gestures depend on the cultural context, but might be intuitive enough to use. On the contrary, we expect the interpretation of pupil movements to massively increase the cognitive load.

Facial expressions.

Pfister reports promising results [9] in automatically detecting and interpreting facial expressions. An expression of confusion could be used to automatically show contextual information. Anger could indicate questionable code quality. Signs of exhaustion could be used to suggest a break.

Blinking.

Triggering an action by blinking is a simply detectable gesture. Blinking is, however, a frequently performed intuitive action. We expect that reacting to such an interaction introduces a high rate of false positives. We also expect a blink-based trigger to confuse and exhaust users.

7. FUTURE WORK

We see our work on gaze-based keyboard focus setting in a programming environment as a first exploration. The main focus of our future work in this area will be the identification of further use cases in programming environments that can be enhanced with gaze tracking information. Glücker [3] concentrates his work on menu and file navigation. We will aim at tasks where the eye takes over less control actions.

Based on our experience, we have claimed that programmers frequently use the keyboard instead of a pointing device to focus a text area. We plan on following this obser-

vation to find out when a pointing device is actually used. That way we expect to identify more use cases where gaze information can be of use.

The situations identified in Section 2 will be in our immediate focus. We will test an environment where information is further away from input fields. We will also concentrate on how to browse lists while viewing on their content.

In order to implement these two use cases, it might be required to treat UI elements differently based on desired behavior. E. g., a list in a browser might have to interpret arrow keys although a user is not looking at it. It is possible that a context-oriented programming approach can be taken.

Our work has raised several questions. The perceived cognitive load described in Section 6.2 needs further investigation. In the context of our implemented framework, the time used to solve a question and the time used to focus it have to be measured independently. Increased cognitive load could account for a longer thinking time, which might have been obscured by a shorter pointing time.

8. CONCLUSION

In this work, we explored techniques that make use of gaze tracking information. Our objective was to apply such techniques in a programming environment.

We concentrated on understanding which element a user intends to manipulate by evaluating his gaze. We described required capabilities of a programming environment that can derive keyboard focus from gaze information. Specifically, we investigated if it is practicable to set the keyboard focus to the screen coordinates of a programmers gaze. We extended a live programming environment accordingly and tested it with a simple task. We found that programmers were able to use gaze for setting the keyboard focus.

We generalized our test setup based on our experiences, showing points where such an environment can be modified in order to compare different approaches. We will use this setup in the future to continue our explorations.

While working in a gaze-enhanced environment, we found that accuracy is very important when controlling aspects of it with gaze. However, even with a very accurate setup, we found hints that gaze control seems to cause fatigue and will elaborate on this topic.

We learned that it is feasible to derive the keyboard focus from view focus in certain situations. For future work, we have set our main objective to evaluating situations where view focus and keyboard focus diverge. Based on our explorations in this area, we expect that the strategies we presented can successfully be adapted in a programming environment.

9. ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of HPI's Research School[3] and the Hasso Plattner Design Thinking Research Program.[4]

[3] www.hpi.uni-potsdam.de/research_school

[4] www.hpi.de/en/research/design-thinking-research-program

10. REFERENCES

[1] H. R. Chennamma and X. Yuan. A Survey on Eye-Gaze Tracking Techniques. Engg Journals Publications, 2013.

[2] A. Clemotte, M. A. Velasco, D. Torricelli, R. Raya, and R. Ceres. Accuracy and Precision of the Tobii X2-30 Eye-Tracking Under Non Ideal Conditions, 2014.

[3] H. Glücker, F. Raab, F. Echtler, and C. Wolff. EyeDE: Gaze-Enhanced Software Development Environments. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, pages 1555–1560. ACM, 2014.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. 32(10):318–326, 1997.

[5] R. J. Jacob. The Use of Eye Movements in Human-Computer Interaction Techniques: What You Look at is What You Get. *Transactions on Information Systems (TOIS)*, 9(2):152–169, 1991.

[6] M. Kumar, A. Paepcke, and T. Winograd. Eyepoint: Practical Pointing and Selection Using Gaze and Keyboard. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, pages 421–430. ACM, 2007.

[7] M. Kumar, T. Winograd, and A. Paepcke. Gaze-Enhanced Scrolling Techniques. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, pages 2531–2536. ACM, 2007.

[8] A. M. Penkar. *Hypertext Navigation with an Eye Gaze Tracker*. PhD thesis, ResearchSpace at Auckland, 2014.

[9] T. Pfister, X. Li, G. Zhao, and M. Pietikäinen. Recognising Spontaneous Facial Micro-Expressions. In *Proceedings of the International Conference on Computer Vision (ICCV)*, pages 1449–1456. IEEE, 2011.

[10] R. Vertegaal. A Fitts Law Comparison of Eye Tracking and Manual Input in the Selection of Visual Targets. In *Proceedings of the International Conference on Multimodal Interfaces (ICMI)*, pages 241–248. ACM, 2008.

[11] X. Xiong, Z. Liu, Q. Cai, and Z. Zhang. Eye Gaze Tracking Using an RGBD Camera: A Comparison With a RGB Solution. In *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 1113–1121. ACM, 2014.

[12] S. Zhai, C. Morimoto, and S. Ihde. Manual and Gaze Input Cascaded (MAGIC) Pointing. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, pages 246–253. ACM, 1999.