iTrace-Visualize: Visualizing Eye-Tracking Data for Software Engineering Studies

Joshua Behler
Department of Computer Science
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

Gino Chiudioni
Department of Computer Science
Kent State University
Kent, Ohio, USA
gchiudio@kent.edu

Alex Ely
Department of Computer Science
Kent State University
Kent, Ohio, USA
aely3@kent.edu

Julia Pangonis
Department of Computer Science
Kent State University
Kent, Ohio, USA
jpangoni@kent.edu

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

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*— iTrace is community infrastructure that allows software engineering researchers to conduct eye-tracking studies on large realistic code bases. The iTrace Infrastructure consists of a set of tools that assist with gathering, processing, and evaluating eye-tracking data on large software projects within an Integrated Development Environment (IDE). A typical eye-tracking study results in millions of raw gazes that are overwhelming to view and sort through. To help researchers view and comprehend this data, iTrace-Visualize is presented. This tool integrates information produced by the iTrace Infrastructure into a dynamic video recording of the eye-tracking session. Eye fixations and the scan path between fixations are overlayed on the video. Additionally, the line being examined can be highlighted in the video. iTrace-Visualize enables a researcher to replay eye fixations via a video overlay immediately after a study. This serves as a quick validation of what was done during the study and can also provide quick insights into what the participants looked at. To illustrate iTrace-Visualize's capabilities, a small pilot study is performed.**
Demo Video—https://youtu.be/c1hUFDmBM50

***Keywords*—**eye tracking, fixations, pipeline

I. Introduction

The iTrace Infrastructure is used by software engineering researchers to perform eye-tracking studies in development environments [1] [2] [3] [4]. Normally, eye-tracking studies are performed on static, unmoving stimuli. Software engineering studies that use eye-tracking have been typically performed on a static snippet of code. This is limiting, as the amount of code a participant can view is only what can fit on a screen. Additionally, participants are viewing the code outside of a normal development environment they use. To address this threat to validity, the iTrace infrastructure provides iTrace-Core and the iTrace IDE Plugins. An IDE Plugin is loaded alongside iTrace-Core to perform an eye-tracking study. iTrace-Core gathers raw gaze data from the eye-tracker, while the IDE Plugin gathers contextual information from the IDE, such as file and the line/column of where the user is looking. The output from iTrace-Core and the IDE Plugins are fed into iTrace-Toolkit. iTrace-Toolkit can then be used to convert the gaze data into eye fixations, and along with a srcML [5] [6] file of the source code, gather contextual information relating to the language context of the source code.

Researchers still face the issue of visualizing the collected eye-tracking data. A five-minute eye-tracking session using a 120Hz eye-tracker generates around 36,000 raw eye gazes and depending on the fixation generation algorithm used in iTrace-Toolkit, hundreds of fixations. To view the data, researchers typically open the files with a database browser, or they output results into a text file and then manually examine them. While information such as the token and context can be understood from these formats, information like the x and y pixel coordinates and the duration are difficult to follow.

Members of the iTrace users community have requested the ability to have a simple way to visualize the information generated by iTrace. To help researchers view the gazes, fixations, and other gathered information, we created iTrace-Visualize. iTrace-Visualize is a tool that combines data gathered from previous steps (typically after the data runs through Toolkit [4]) and marks up a video to display the data concisely and simply. iTrace-Visualize offers the following types of markups:

- **Gaze Markup:** Using the gaze and IDE context data gathered from iTrace-Core and an IDE Plugin, the gazes are displayed on the video when they occur in real-time.
- **Fixation Markup:** Like the gaze markup, fixations generated in iTrace-Toolkit can be displayed. Specific fixation runs can be chosen if multiple are run and available in Toolkit. Several fixation algorithms are supported.
- **Saccade Markup:** Saccades, the path the eye takes between fixations, are calculated and drawn to the video.
- **Code Highlighting:** By putting frames of the video through an image processing pipeline, each line of code is

detected and given a bounding box. iTrace-Visualize then highlights the line when a fixation is within the bounds.

- **Video Interpolation Stretching:** If a high-speed eye-tracker is used during the eye-tracking session, and the refresh rate is much higher than the FPS of the recorded video, iTrace-Visualize can duplicate the frames of the video to stretch it out, allowing for more data to be displayed.
- **Fading Display:** Due to the instantaneous nature of gazes, displayed gazes are displayed on a single frame before being replaced by the next gaze. To solve this, a fading display is implemented to slowly fade away old gazes so instead of one single gaze point flitting around the video, a cloud of gaze points shifts around the screen.
- **Options Customization:** Multiple options within iTrace-Visualize can be customized and changed, granting researchers the ability to display information they want in a way they want.

Trace-Visualize is currently the last step of the iTrace Infrastructure, as it makes use of information from almost every previous step in the process. Figure 1 details where iTrace-Visualize lies within the greater iTrace Infrastructure.

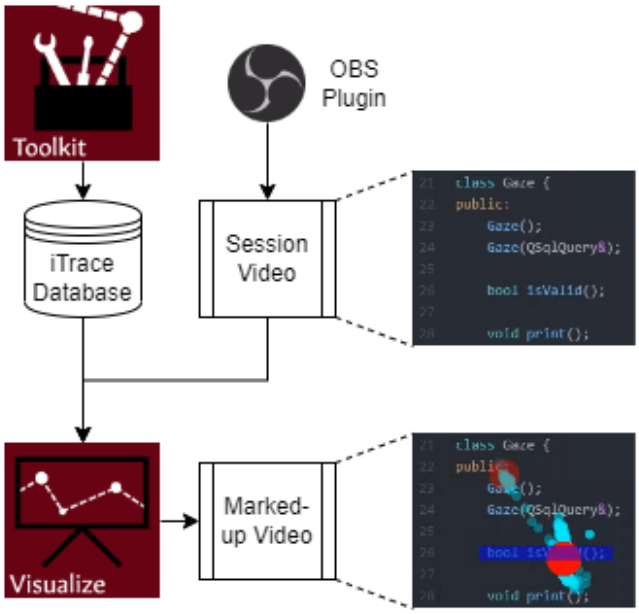

Figure 1: iTrace-Visualize within the iTrace Infrastructure and its input/output

II. Related Studies

Previous work has been done on how to best visualize eye-tracking data. Špakov and Miniotas did work on visualizing the data as a heat map over the source [7]. Similar studies by Punde, *et al.* [8] and Pfeffer and Memili [9] have been done using heatmaps. This method was explored for use within iTrace-Visualize, but it was ultimately decided that a heat map does not fit the dynamic nature of a software engineering study, due to the large amount of color and detail on the stimulus, and the time progressing nature of the eye-tracking data. Additionally, because iTrace-Visualize has different data to mark up (saccades, gazes, fixations), using heatmaps becomes complicated and difficult to read. We prioritized fixations/saccades over heatmaps as they are more focused on what was looked at by the individual person. Other adhoc visualizations using graph embeddings [10] are proposed by Zhang et al. however these are seen as more specific to a task and not real-time like iTrace-Visualize.

Previously within iTrace we explored visualizing gazes as they are being recorded. iTrace Eclipse, introduced by Sharif and Maletic, has a feature that highlights any token that falls under the gaze within the IDE [1] [2]. Work has also been done by Clark and Sharif on iTraceVis [11], an early feature in iTrace-Eclipse which did live visualization directly in Eclipse. These features are what inspired us to build iTrace-Visualize. Using it allows a researcher to go from conducting the study to visualizing the data for quick insights. It was also the most requested feature from the community.

III. Architecture

A. *Implementation*

iTrace-Visualize is implemented using Python and the QT Python Bindings [12]. iTrace-Visualize consists of a simple GUI that consists mostly of buttons for importing data and customizing output options. The OpenCV Python bindings [13] are used for line detection.

To display markup, iTrace-Visualize creates a list of timestamps that correspond to every frame of the video and gathers a list of every element of data that the researcher wants to be displayed. Every list is then looped through, and timestamps are compared. If the timestamp of a particular data point matches the current frame, the data is drawn.

B. *Video Gathering*

For iTrace-Visualize to markup a video, a video recording must be taken of the eye-tracking session. However, because of how iTrace-Visualize draws markup based on the timestamp of the data, making sure the video's start and end matches perfectly with the start and end of the eye-tracking session is very important. Manually taking a recording of the session can introduce slight timing offsets, which results in the whole output being off.

To solve this problem, a plugin was developed for the OBS program, called iTrace-ScreenRecord. OBS [14] (Open Broadcaster Software) is a program designed for the recording and streaming of desktop programs. A researcher can set OBS to record the IDE before a session and connect OBS to iTrace-Core using iTrace-ScreenRecord. After connecting, OBS will automatically trigger a local recording to start when the "Start Tracking" button is pressed in iTrace-Core, and the recording will finish and save when the session is ending. This recorded video will be approximately the same length as the session, and thus does not cause an offset or cutoff in the markup. Using OBS to record this video provides the advantages of allowing a researcher to record desktop and microphone audio, record the entire desktop or specific programs, and all the other features that OBS provides.

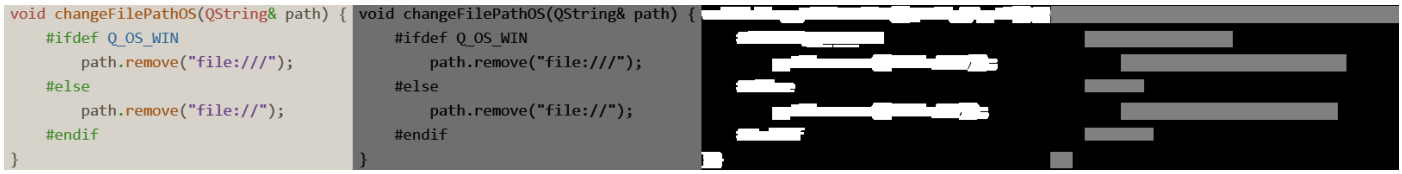

Figure 2: From left to right – an input image (inverted), the darkened grayscale image, the image dilation, and the lines' bounding boxes.

C. Gaze and Fixation Markup

Gaze data is relatively simple to draw for markup. Gazes have no duration, and only have an (x, y) pixel coordinate value and a timestamp. Gazes are selected per recording session and can be chosen from a selection menu within iTrace-Visualize. Gazes are drawn as a simple five-pixel radius circle on the video.

Fixations can be selected after choosing a recording session and are drawn in a similar way to gazes, as a circle. However, fixations have a duration, and so are drawn onto the screen for more than a single frame. Fixations are drawn onto the screen for as many milliseconds as their duration. Fixations also grow over their duration, so longer fixations end up being larger than shorter ones.

D. Saccade Markup

Currently, saccades are not calculated during any previous step within the iTrace Infrastructure. It is intended for iTrace-Toolkit to eventually do this, but for now iTrace-Visualize calculates the saccades. iTrace-Visualize defines a saccade as a grouping of gazes that occur between two other fixations but are not a part of said fixations. Saccades are drawn as a series of white lines drawn between gaze points. While they do not have a predefined duration like fixations, they will be drawn on screen while any of its gazes are valid.

E. Code Highlighting

Along with marking up the concrete eye-tracking data, iTrace-Visualize provides an optional feature to highlight the line of code that the user was looking at during a fixation. This information gives researchers a visual way to see what coding constructs a participant looks at while analyzing code.

Before iTrace-Visualize highlights the line, each frame of the video must be prepared so that the bounding boxes of each line are determined. To do so, the frame is put through a couple steps, some of which are seen in Figure 2:

1. If the frame is from an IDE using a dark mode theme, the image must be inverted first, so the frame is similar to a light mode theme.
2. The image is converted to grayscale, and then darkened. This causes any changes in font color, which is common in IDE's due to syntax highlighting, to become evened out.
3. The image is dilated, with an emphasis on horizontal dilation. The horizontal dilation causes characters and words on a line to merge into each other, while lines stay separated.
4. For each blur in the dilated image, a flood fill algorithm is used to find the bounding box coordinates of the blur.

The bounding box coordinates gathered at the end of the process are then used for both calculating if a fixation is looking at a particular line, as well as providing the dimensions for which pixels to affect to highlight the line.

A major issue with implementing the highlighting is how long identifying each bounding box can take, especially considering the efficiency of the flood fill algorithm. If each frame is individually calculated, the process will take hours. To solve this, iTrace-Visualize calculates the bounding boxes once, and reuse them until the scene drastically changes. Every frame is compared to the previous one, and if the percentage of change is over five percent, the previous boxes are discarded, and a new set is calculated.

F. Video Interpolation Stretching

Most current eye-trackers available to researchers have refresh rates higher than 60Hz. This means that any screen recording made of a session using these eye-tracker must be recorded at a higher FPS to prevent gazes from being skipped during visualization. Most machines and a lot of monitors cannot record and display video consistently at framerates higher than 144Hz. The fading display helps combat this by displaying any skipped gazes as already fading, but the multiple gazes still appear at once. To help combat this issue, iTrace-Visualize artificially stretches a video out by adding duplicate input frames to be processed. Each duplicated frame is added to the input queue and given an interpolated timestamp. These frames are treated as normal input, and any data that aligns with the frame's timestamp are drawn on as normal.

Videos can be extended by any integer factor, with a factor of N adding N duplicates of each frame. The output videos is N times longer than the input video, as the output will always be the same framerate as the input. This has the side effect of causing the video to appear to be running in slow motion, and no longer be in real time.

G. Fading Display

Gazes are an instantaneous data point, only appearing on the frame closest to their timestamp. Because of this, when viewing a video, a single gaze dot will jump around the screen every frame and is hard and disorienting to follow. To remedy this, a fading approach to displaying data is adopted. Instead of drawing only the most current data, all data points that occurred in the past within a customizable period are drawn. By default, iTrace-Visualize draws everything within a one second

window. These markups slowly fade out, to prevent crowding of the data. Figure 4 details the change in display methods.

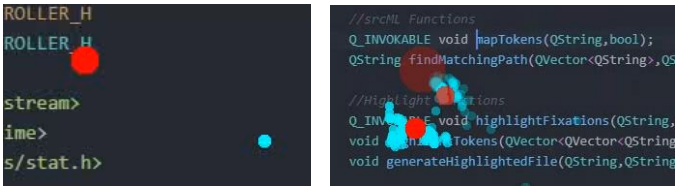

Figure 4: A frame of marked video without fading (left) and with fading (right) shown on different snippets.

H. Options Customization

To grant researchers the ability to fine tune their visualized data, iTrace-Visualize allows a researcher to tweak various values and options for their output.

- **Fixation Drawing:** If a researcher does not want fixations to be displayed, they can avoid selecting a fixation run in the top-right list of runs after selecting a session.
- **Saccade Drawing:** A checkbox is provided to enable and disable the selection of saccades.
- **Line Highlighting:** Like saccades, a checkbox is provided to enable and disable the highlighting of lines. This feature also is not performed if fixations are not selected.
- **Value Tweaking:** Gaze size, fixation base size, video stretch factor, and the fade delay can be manually adjusted through a number text box.

IV. Results

To test iTrace-Visualize, a small scale study is conducted with four members of the iTrace development team. To begin, a normal eye-tracking session is set up using iTrace-Core (with DejaVu [3] enabled via a checkbox) and whatever IDE the participant preferred. For this test, we used the iTrace-Atom plugin. We used the DejaVu option to keep track of the mouse scrolls and accurately record data over 60 Hz [3]. Recording with DejaVu is not required to use iTrace-Visualize. The participants are given a small part of iTrace-Toolkit's source code, and asked to read the function name in every `.h` file and read through a random function of their choice in the `controller.cpp` file. Participants are instructed to set up OBS and iTrace-ScreenRecord during the recording as well. After recording, participants are instructed to run the source code through srcML, and then use iTrace-Toolkit to map tokens and generate a set of fixations with every algorithm.

After gathering all the data and videos from the participants, the first author ran each session through iTrace-Visualize. The output videos are collected and watched through, and compared to standard heatmap style images from the heat map studies mentioned in Section II [7] [8] [9]. Table 1 details the sessions and how long they took to process. For consistency, only fixations calculated with the IDT algorithm [15] in iTrace-Toolkit are displayed and counted.

Table 1: Sessions gathered and put through iTrace-Visualize. All time values are in hh:mm:ss

Participant	1	2	3	4
# of Gazes	40519	30453	35766	78594
# of Fixations (IDT)	254	211	1273	488
# of Mouse Scrolls	70	114	368	108
Session Time	0:02:15	0:01:41	0:04:57	0:04:21
Proc. Time w/o Highlighting	0:31:24	0:22:46	0:30:44	0:59:40
Proc. Time w/ Highlighting	0:42:12	0:37:55	1:21:09	1:35:15

Participants 1, 2, and 4 used the Tobii Pro Spectrum eye-tracker at 300Hz, while participant 3 used the Tobii Pro X3-120 eye-tracker at 120 Hz. Because participant 3 used a lower speed tracker, they have significantly less gazes for the amount of time recorded (around 5 mins). This also affects the number of fixations, as participant 3's gazes are less dense, causing more to be registered. Despite this, participant 3 has similar number of gazes as participants 1 and 2, and thus has a similar processing time when not doing line highlighting. Participant 4, has almost double the number of gazes and double the processing time.

Line highlighting affects processing time differently. Because the bounding boxes must be re-calculated when the screen signifigantly shifts, things like scrolling the screen or opening a context menu causes delays in processing. Participant 3 has the largest number of mouse scrolls, which results in the largest difference in processing time when processing without highlights compared to with highlights. This does not account for every increase in processing time, as other things like context menus and pop-ups also increase the time.

V. Discussions

When comparing the two styles of visualizations, there is not a lot of objective pros and cons that can be listed, as preferences may be based on subjective choice. However, iTrace-Visualize's implementation does provide some advtanages over the more traditional heatmap visualizations. The biggest improvement is the ability to view the data over time, and tell the order and duration of what a user looking at. While heatmaps can showcase the progression of time, changes in heatmaps take longer to update, as they are area based as opposed to iTrace-Visualize's point-based system. Heatmaps also do not scale well with the differing speeds of eye-trackers and different fixation algorithms. Because different numbers of gazes can go into each fixation due to these factors, the heatmaps do not consistenly grow for each session, and some appear weaker. Additionally, iTrace-Visualize supports the drawing of multiple kinds of data (saccades, raw gaze, and fixations) that are not seen on heatmaps.

iTrace-Visualize will take a good chunk of time to process a video as shown in Table 1. Even before the introduction of the fading display and line highlighting, iTrace-Visualize must draw markup on every single frame of a video, which at two

minutes and 60 fps is 7200 frames. Additionally, because of the fading display, multiple data points must be drawn on each frame. At 300Hz, roughly 300 gazes are drawn on each frame at varying levels of transparency. The average time spent on each frame is calculated in Table 2. For the 3 sessions recorded at 300Hz, the average processing time without highlighting is 0.2283 seconds per frame. This is consistent regardless of video length.

Table 2: Average processing time per frame in iTrace-Visualize. All time values are in seconds.

Participant	1	2	3	4
Tracker Speed	300Hz	300Hz	120Hz	300Hz
# of frames	8101	6081	17861	15709
Proc. Time w/o Highlight	1884	1366	1844	3580
Avg. Frame Time w/o Highlight	0.2325	0.2246	0.1032	0.2278
Proc. Time w/ Highlight	2532	2275	4869	5715
Avg. Frame Time w/ Highlight	0.3125	0.3741	0.2726	0.3638
% Increase in Frame Time	134.3%	166.5%	264.0%	159.6%

When using highlighting, the scale is different. The percent increase in frame processing time does not stay consistent due to the number of bounding box calculations that must be performed. Out of the three participants at 300Hz, participant 2 has the biggest percent increase, with 4 and 1 behind in order. This matches with the number of mouse scrolls in Table 1. This can also be seen with Participant 3, because despite having a much lower eye-tracker speed, there is an enormous increase in frame processing time due to the larger amount of bounding box refreshes that had to be performed.

VI. Conclusions and Future Work

iTrace-Visualize takes eye movement data from the iTrace pipeline and marks up the data onto a video recording of an eye-tracking session for other researchers to view and present their data. This is helpful for researchers so they can visually analyze their collected data for flaws and inconsistencies or compare data in a human-perceivable way.

In the future, we plan to expand iTrace-Visualize by expanding the type of markup iTrace-Visualize can support. Mouse and keyboard information gathered from DejaVu [3] in iTrace-Core can be displayed through small visual annotations for keyboard presses and expanding/shrinking circles for mouse clicks. When the iTrace Infrastructure supports the calculation of saccades, iTrace-Visualize will be updated to use those instead of manually calculating them from fixations.

Highlighting will also be improved, with both line and token highlighting being offered. Currently, highlighting will only occur if a fixation occurs directly in the bounding box of a line. However, highlighting should occur if the fixation is close to the box as well, as the box will wrap the text to its pixel values, which does not necessarily match the dimensions of the line.

Additionally, the processing speed for iTrace-Visualize will be improved. When dealing with numerous lengthy studies, the ability to generate a visualization of each session quickly is important. Allowing iTrace-Visualize to process the video in multiple threads will increase the speed. iTrace-Visualize can also be improved to do its additions on the GPU of the machine, if present, which will drastically speed up the process.

We plan to release a beta version of iTrace-Visualize to the public by the end of the summer.

References

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

[2] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "itrace: Eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ACM, 2018, p. 105.

[3] V. Zyrianov *et al.*, "Deja Vu: semantics-aware recording and replay of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks—methodology and analyses," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 168, Dec. 2022, doi: 10.1007/s10664-022-10209-3.

[4] J. Behler, P. Weston, D. T. Guarnera, B. Sharif, and J. I. Maletic, "iTrace-Toolkit: A Pipeline for Analyzing Eye-Tracking Data of Software Engineering Studies," presented at the in the Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE) Demonstrations Track, Melbourne, Australia, May 2023.

[5] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2011, pp. 173–184. doi: 10.1109/SCAM.2011.19.

[6] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *29th IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 516–519. doi: 10.1109/ICSM.2013.85.

[7] O. Špakov and D. Miniotas, "Visualization of Eye Gaze Data using Heat Maps," *Elektron. Ir Elektrotechnika*, vol. 74 No. 2, pp. 55–58.

[8] P. A. Punde, M. E. Jadhav, and R. R. Manza, "A study of eye tracking technology and its applications," in *2017 1st International Conference on Intelligent Systems and Information Management (ICISIM)*, Oct. 2017, pp. 86–90. doi: 10.1109/ICISIM.2017.8122153.

[9] T. Pfeiffer and C. Memili, "Model-based real-time visualization of realistic three-dimensional heat maps for mobile eye tracking and eye tracking in virtual reality," in *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications*, in ETRA '16. New York, NY, USA: Association for Computing Machinery, Mar. 2016, pp. 95–102. doi: 10.1145/2857491.2857541.

[10] L. Zhang, J. Sun, C. Peterson, B. Sharif, and H. Yu, "Exploring Eye Tracking Data on Source Code via Dual Space Analysis," in *2019 Working Conference on Software Visualization (VISSOFT)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 67–77. doi: 10.1109/VISSOFT.2019.00016.

[11] B. Clark and B. Sharif, "iTraceVis: Visualizing Eye Movement Data Within Eclipse," in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, Shanghai, China: IEEE, Sep. 2017, pp. 22–32. doi: 10.1109/VISSOFT.2017.30.

[12] "Qt for Python." https://doc.qt.io/qtforpython-6/ (accessed Jun. 21, 2023).

[13] "opencv-python · PyPI." https://pypi.org/project/opencv-python/ (accessed Jun. 21, 2023).

[14] "Open Broadcaster Software | OBS." https://obsproject.com/ (accessed Jun. 21, 2023).

[15] R. Andersson, L. Larsson, K. Holmqvist, M. Stridh, and M. Nyström, "One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms," *Behav. Res. Methods*, vol. 49, no. 2, pp. 616–637, Apr. 2017, doi: 10.3758/s13428-016-0738-9.