Eye Tracking in Computing Education

Teresa Busjahn
Freie Universität Berlin
busjahn@inf.fu-berlin.de

Carsten Schulte
Freie Universität Berlin
schulte@inf.fu-berlin.de

Bonita Sharif
Youngstown State University
bsharif@ysu.edu

Simon
University of Newcastle
simon@newcastle.edu.au

Andrew Begel
Microsoft Research
abegel@microsoft.com

Michael Hansen
Indiana University
mihansen@indiana.edu

Roman Bednarik
University of Eastern Finland
roman.bednarik@uef.fi

Paul Orlov
University of Eastern Finland
paul.a.orlov@gmail.com

Petri Ihantola
Aalto University
petri.ihantola@aalto.fi

Galina Shchekotova
JetBrains
gshchekotova@gmail.com

Maria Antropova
JetBrains
maria.antropova@gmail.com

ABSTRACT

The methodology of eye tracking has been gradually making its way into various fields of science, assisted by the diminishing cost of the associated technology. In an international collaboration to open up the prospect of eye movement research for programming educators, we present a case study on program comprehension and preliminary analyses together with some useful tools.

The main contributions of this paper are (1) an introduction to eye tracking to study programmers; (2) an approach that can help elucidate how novices learn to read and understand programs and to identify improvements to teaching and tools; (3) a consideration of data analysis methods and challenges, along with tools to address them; and (4) some larger computing education questions that can be addressed (or revisited) in the context of eye tracking.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Experimentation, Human Factors, Measurement

Keywords

CS Ed Research; Code reading; Computing education; Empirical research; Eye tracking; Gaze analysis; Program comprehension; Programming education; Teaching programming

ICER'14, August 11-13, 2014, Glasgow, Scotland, UK.

http://dx.doi.org/10.1145/2632320.2632344.

1. INTRODUCTION

This paper introduces eye tracking as an instrument for computer science education research. The approach builds on the outcomes of the 1st International Workshop on Eye Movements in Programming Education: Analyzing the Expert's Gaze [4], held in conjunction with the 13th Koli Calling International Conference in Computing Education Research [15]. The workshop brought together educators and practitioners to analyze how eye tracking and the rich data it affords could benefit programming education.

The observation of eye movements adds an objective source of information about programmer behavior to the collection of research methods in computing education which can be used to facilitate the teaching and learning of programming. Alternative approaches for gaining insights about learners' programming processes include analyzing the consecutive versions of assignments submitted for automated assessment [1], instrumenting student programming environments to record snapshots from compilation [18], and recording keyboard events in text editors [11]. All of these methods complement one another; augmenting them with eye tracking would provide a comprehensive view of the process of learning programming. While our focus in this paper is on programming education, eye tracking is a valuable instrument for other areas of computing education as well, *e.g.* understanding graphical data models [17].

This paper is organized as follows, the next section gives an overview of eye tracking technology, and presents previous work. In section 3, we describe our case study and the eye movement coding scheme. Finally, to address challenges we discovered in our study, we offer a set of ideas and tools to advance eye tracking in computing education. We conclude with an outline of ways in which eye tracking can help answer crucial questions in computing education.

2. GAZE IN COMPUTING

2.1 A Brief Introduction to Eye Tracking

Eye trackers are used to capture a user's eye movements when he looks at a stimulus while working on a task. In

computing education, a programmer would be given a programming task to solve while looking at relevant source code. There are two important types of eye movements: the *fixation*, which is the settling of the eye gaze on an object of interest for a minimum period of time, and the *saccade*, which is a quick movement of the eyes from one location to another. Both fixations and saccades are voluntary, and indicate the location of the subject's attention. A *scan path* is a directed path formed by saccades between fixations. Processing of visual information occurs only during fixations [14], *i.e.*, as a programmer looks at programming constructs in code, various mental processes are triggered to solve the task at hand.

Eye tracking is a source of rich and valuable information which cannot be obtained by other methods. Conventional measures retrospectively record the accuracy of the subject's response and the time taken to obtain that response. For example, a programming educator will ask students to report their answers after debugging or tracing a program in a lab. This method records only the final outcome after the specific task has ended, neglecting information that might help understand how and why a student chose a particular (correct or incorrect) answer. Additionally, these measures raise a potential threat to the validity of the task, namely the difference between student responses upon completion of a task and the reality the student experienced while performing that task. In other words, a student may misreport an experience at the end of a long task, or may forget to report it altogether.

Researchers can address this issue by asking programmers to record their observations while working towards their answers. However, this has the risk of interrupting their work on the main task at hand. This same drawback exists even if explicit methods such as think-aloud are used, since they still distract the programmers from their core task. Moreover, subjects must be constantly reminded to verbalize their thoughts, since they do not often do so while they program in their natural setting. Even expert programmers find it difficult to state out loud exactly how they read a program. Many unconscious decisions go unreported, for example, encountering logical dead ends while reading a program.

Much of the critical information that is lost with traditional methods of measurement and assessment can be recovered using an eye tracker. There is nothing the programmer needs to wear in order for their eye movements to be recorded. Modern state-of-the-art eye trackers consist of a small hardware device that is placed near the programmer's monitor and can silently and unobtrusively document eye movements while the programmers looks at the screen. The additional data provides insights into the programmer's thought processes, and achieves a finer granularity of data capture across space (across the program) and time (as the task progresses) because tacit knowledge and understanding is made more explicit.

Furthermore, eye tracking makes it possible to take advantages of verbal accounts without the drawback of imposing additional cognitive load and interfering with the comprehension process at hand. Combining retrospective think-aloud with eye tracking, study participants initially work purely on their task (e.g. understanding source code). Once the task is complete, they are prompted to verbalize their thoughts, watching their recorded eye movements to aid their recollection [24, 13]. This technique makes complementary use of think-aloud and eye tracking and has been found to induce higher quality comments about cognitive processes. We strongly believe that eye tracking synergizes with other methods of assessing comprehension, and used together provides additional insights.

2.2 Previous Work on Eye Tracking in Computer Programming

Eye tracking has been studied in non-computer fields such as chess, reading, piloting [12], mammography [21], and surgery [25]. Here, we present an overview of work that has used eye tracking in programming research.

Crosby and Stelovsky [9] were pioneers of using eye tracking to study programmers. They found that programmers employed several distinct types of scan path patterns while they read an algorithm written in Pascal.

Uwano et al. [24] studied eye gaze patterns while five programmers detected code defects. They identified a pattern called *scan*, in which programmers appear to form an overview of the code. Approximately 70% of source code lines were viewed in the first 30% of the time spent reading the code. Sharif et al. [20] replicated this experiment with a larger sample of 15 programmers and found similar results. Programmers who spent less time to initially scan the code tended to take more time to find defects.

Fan [10] analyzed the eye gaze of programmers to learn about program comprehension processes used for beacons and comments in different tasks. Code scanning sequences were directly affected by comments, enabling programmers to chunk larger code blocks. Fan concluded that eye gaze data is very useful in documenting and analyzing the program comprehension processes.

Busjahn et al. [7] used eye tracking to compare natural language text reading and code reading. They found a significant increase in both fixation duration and number of backward movements when subjects read source code, indicating the different demands of these two text types and the reading patterns that they induce.

Bednarik [3] studied the differences between novices and experts during debugging using source code and graphical representations. He found that repetitive eye patterns were associated with less expertise; novices used both representations with a lot of context switching.

Turner et al. [23] conducted a preliminary study on 38 students to assess how the choice of programming language affected how programmers solve tasks. Looking at simple C++ and Python programs, they found a significant difference between the two languages for the fixation rate on buggy lines of code.

There have been a few studies using eye tracking to study programming, but none comprehensively analyze the relationship between raw eye movements and comprehension. It is extremely difficult to translate a person's eye movements into insights about his or her mental state while reading and understanding program [6]. We are confident that there will be more of these studies because of the diminishing cost of eye tracking. One of our goals is to raise awareness of the opportunities and challenges afforded by this technology in computing education research.

3. CASE STUDY

We conducted a case study to determine whether the use of eye tracking in computing education was feasible, could provide rich data for analysis, and could lead to novel teach-

ing ideas. For details beyond those in this paper, please read our technical workshop report [4].

3.1 Experimental Setup

The eye movement data analyzed for the workshop came from a study with professional software developers reading and understanding short Java programs. We recorded them in an office at the programmers' company with an SMI REDm 120 Hz eye tracker using the OGAMA tracking software.[1]

The recording sessions started with natural language texts followed by comprehension questions to familiarize the subjects with the instrument and the tasks. The subjects then moved on to examine Java code. After being informed that the code did not contain bugs, they were asked to read it, comprehend it, and answer a question to test their comprehension. The code segments were short enough to fit on a single screen without scrolling, to simplify the connection between gaze location on the screen and in the code.

The program read by the subjects (shown in Figure 1) calculated the area of a rectangle. Subject 1 was told to expect to answer a question about the return value of the `rect2.area()` method, while Subject 2 was told to expect a multiple-choice question about the algorithm in the code.

Each trace was given to the workshop participants as an AVI video showing the subject's current fixation location as a large red circle on top of the source code that the subject saw.[2] The prior five fixations were marked by blue circles whose size indicated the duration of the fixation. Blue lines joining the circles represented the saccades. The eye movements were quite rapid (and usually are), so the researchers could slow down the video to see all the gaze locations.

The two traces are very different. One might consider subject 1's gaze to be erratic. Viewed in real time, it flashes wildly about the code, generally spending very little time on any one point. However, when taken in total, there is a clear pattern of subject 1's eye returning to certain focal points. These points are pertinent to the question the subject was told to expect, but the fixations are so brief as to leave the analyst wondering whether if it was at all possible to gain any comprehension of the code. For example, in one 10 second span, the gaze shifts more than a dozen times between every method on the screen, typically spending less than a second on each point of interest.

By contrast, Subject 2 reads the code slowly and methodically, yet takes about 40% less time overall than Subject 1. We see evidence of linear scanning patterns, and very long gaze fixations on areas of interest. Whereas Subject 1 spent 10 seconds looking at every method on the screen, Subject 2 spent 10 seconds looking at a single variable declaration. In addition, after 1 second glances at the methods `height()` and `width()`, there was a steady 4 second gaze on `area()`, followed by another 8 seconds on the declaration of `rect2`. Our impression was that Subject 2 deliberately read through the code, understanding it the first time it was viewed.

3.2 The Workshop

We designed the Eye Movements in Programming workshop to bring together a number of researchers to consider various approaches of inferring cognitive processes from eye

[1] http://www.ogama.net

[2] These videos are available at http://www.mi.fu-berlin.de/en/inf/groups/ag-ddi/Gaze_Workshop/koli_ws_material.

```
1 public class Rectangle {
2     private int x1 , y1 , x2 , y2 ;
3
4     public Rectangle ( int x1 , int y1 , int x2 , int y2 ) {
5         this.x1 = x1 ;
6         this.y1 = y1 ;
7         this.x2 = x2 ;
8         this.y2 = y2 ;
9     }
10
11    public int width ( ) { return this.x2 - this.x1 ; }
12
13    public int height ( ) { return this.y2 - this.y1 ; }
14
15    public double area ( ) { return this.width ( ) * this.height ( ) ; }
16
17    public static void main ( String [ ] args ) {
18        Rectangle rect1 = new Rectangle ( 0 , 0 , 10 , 10 ) ;
19        System.out.println ( rect1.area ( ) ) ;
20        Rectangle rect2 = new Rectangle ( 5 , 5 , 10 , 10 ) ;
21        System.out.println ( rect2.area ( ) ) ;
22    }
23 }
```

Figure 1: Source code used for the workshop – overlaid with eye movements

movements during source code reading. Prior to the workshop, the participants were given access to the two gaze traces described above. The workshop organizers developed a coding scheme for the gaze traces based specifically on the two eye gaze trace videos and the specific Java program, in order to broaden our knowledge of program comprehension strategies. We made a fundamental decision to distinguish between objective and subjective behaviors. At the most basic level, analysts using the scheme would objectively code the part of the program on which the programmer's gaze is resting. At the next level, they would use subjective codes to describe their inferences of the patterns of eye movement and the strategies being employed by the programmer to comprehend the code.

Each workshop participant individually analyzed the eye movement records (without any audio) and coded it using the ELAN video annotation software.[3] They each then wrote a position paper to describe the traces, to reflect on the validity and utility of our coding scheme, and to discuss possible applications of eye movement research for computer science education. At the full day workshop, participants explored their findings with one another, using their discussions to refine the coding scheme and plan further research.

3.3 Coding Scheme

The coding scheme used in the case study captures the objective eye tracking data as well as the coder's inferences about the programmer's comprehension of the source code. We present the coding scheme here to illustrate the possible outcomes of employing eye tracking in computing; it is not

[3] http://tla.mpi.nl/tools/tla-tools/elan

directly generalizable and should not be taken as a central contribution by itself.

We revised the coding scheme according to suggestions given by the participants in their position papers, and further refined it during the workshop (see report [4]). The scheme consists of a number of 'tiers', each of which can be coded with a choice of values. The tiers are summarized below.

Line: indicates the line of code the participant's gaze is on.

Block: indicates the block of code that the line is in. In a typical short code segment, the blocks might be the class `Attributes`, the `Constructor`, the `Main` method, or any other method.

SubBlock: some blocks have identifiable sub-blocks in which a reader's gaze might rest. For example, a method may contain sub-blocks of `Signature`, `Body`, `Method Call`, and `Return`. While the latter two statements are part of the method body, we code them separately because their importance makes them likely to be the focus of the reader's concentration.

Signature: when the gaze rests on the signature sub-block, this tier further indicates whether it dwells on the method `Name`, its `Return Type`, or its `Formal Parameter List`.

Method Call: when the gaze rests on a method call, this tier is used to indicate whether it focuses on the method's `Name` or its `Actual Parameter List`.

These first five tiers refer to the gaze location at a single point in time. They can be coded objectively and automatically based simply on the eye gaze location on the screen.

The next tier of the coding scheme, **Pattern**, identifies particular combinations of the observed fixations. For example, in the pattern we call `Flicking`, the gaze flicks back and forth between two (or possibly more) identified gaze points. Specific instances of this pattern may flick between the actual and formal parameter lists of a method call and its declaration, between the use and declaration of a variable, or between different locations where a variable is used. Other patterns include `Linear Scan`, in which the gaze moves linearly through some part of the code; `Jump Control`, in which gaze follows the code in execution order; and `Thrashing`, in which the gaze leaps about all over the code with no discernible intent. To date, we have identified 11 patterns. While patterns are observable in the eye gaze trace, coders must make a subjective decision about the number of fixations to combine into a single pattern; thus, these patterns cannot be automatically identified without human intervention.

The final tier of the coding scheme, **Strategy**, relies on interpretation by the coder. The analyst uses these codes to determine the cognitive actions taken by a programmer comprehending the program. Some strategies tend to be associated with particular patterns, but there is no one-to-one relation between them. For example, the `Design at Once` strategy is often associated with a linear pattern and suggests the programmer is reading sequentially through part, or all of the code, to acquire an overall understanding. `Intraprocedural` and `Interprocedural Control Flow` follow the expected control flow of the program, which implies that the programmers is simulating program execution. `Test Hypothesis` involves repetition of a gaze pattern, suggesting increased concentration is needed to better understand a particular detail of the program. `Trial and Error` is essentially a `Linear Scan` pattern with faster reading, irregular jumps, and repetition. This code is used when the programmer is searching for some part of the code that will lead to an initial understanding. So far, we have identified 14 distinct strategies.

3.4 Interpretation of Results

Our analyses of the two traces proved extremely interesting. Subject 1, whose gaze we described as erratic, correctly answered the study question. Subject 2, whose gaze seemed to be more methodical, chose the wrong answer to a multiple-choice question about the code. Since both of these subjects were expert programmers, we think it is unlikely that the differences in the accuracy of their answers are directly related to the differences in their gaze patterns. Indeed, we would hope that Subject 2's wrong answer indicated a simple slip, rather than a failure to comprehend the code. Perhaps Subject 1 got his answer correct because his task was more specific than Subject 2's task. Subject 2 had to memorize more of the code and remember four specific variables (`x1`, `x2`, `y1`, and `y2`) in order to choose the correct answer. Nevertheless, it is clear that different experts can display entirely different gaze patterns while reading the same code for comprehension. This diversity was apparent in the traces, even though the program was very short, simple, and bug-free.

3.5 Lessons Learned

As we expected, we had to revise and refine the coding scheme during the course of the case study. Participants found using ELAN to code the low-level categories (*e.g.* `Block`) to be time-consuming and inconsistent, and wished for a tool to automate this step.

The distinction between patterns as objective, observable behavior and strategies as the associated cognitive processes was valuable. However, the coding process is necessarily subjective and the coders could not be definitive about the readers' cognitive processes. Perhaps at this early stage of the research, it would be beneficial to complement the eye movement analysis with other methods, such as retrospective think-alouds.

Our aim is to correlate the subjects' cognitive strategies with observable patterns, so that we might use the pattern to identify the strategy being applied. This would make strategy coding less subjective, but is going to require a great deal more analytical work before it becomes feasible.

One threat to the validity of our coding scheme is that we based it on just two expert gaze traces of one program. However, we believe that our collaborative experiment has helped us establish a foundation that supports additional data analysis and the elaboration and development of more sophisticated analysis methods and materials. Future studies that vary programs, problem domains, and test subjects will enable the research community to refine and improve our expanded understanding of the cognitive processes involved in program comprehension.

4. DATA ANALYSIS - CHALLENGES AND SOLUTIONS

In this section, we discuss the challenges raised in the case study and explain how we addressed them. We present several tools to support interpretation of eye movement data and of records annotated with the coding scheme. Even

though the interpretation of eye gaze data is not entirely straightforward, our workshop made it clear to us that the main challenges are already well understood.

Eye tracking videos are useful for spot-checking specific points in an experiment (e.g., did the subject look in region X at time T), but it is not easy to get a big picture sense of the subject's behavior or to compare it with other subjects' behaviors. Static visualizations like heatmaps and fixation scatter plots can provide such global pictures, but these fail to capture the dynamics and nuances of subject behavior. Accurate eye tracking data interpretation requires additional tools and methods to combine the various views that arise during the analysis.

4.1 Visualizing Annotated Gaze Records

Here we present several tools to help interpret data annotated with ELAN-assigned codes. First, we present an eye movement flow chart in Figure 2, made using the D3.js library. This flow chart represents a graphical Markov chain of the elements in the coding scheme. Flow charts can help analysts find and understand eye gaze patterns, and can also be used in exploratory analysis.

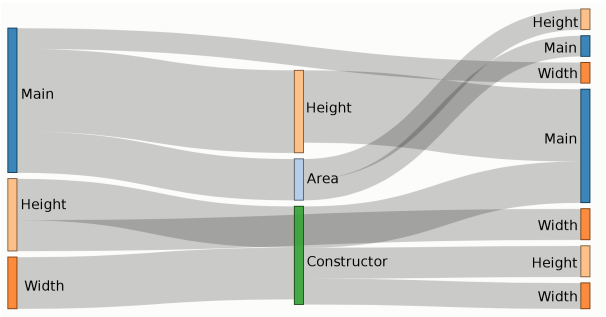

Figure 2: Gaze transition flow chart for Subject 1

The curved lines in the chart represent transitions of the subject's eye gaze from a source (left) to a target (right). The chart in Figure 2 shows two consecutive transitions. The width of the curved lines indicates the fraction of times (*i.e.*, probability) the subject transitioned from the particular source to that particular target. For example, Subject 1 looked at `Main` and then `Height` about twice as often as he did from `Main` to `Area`. However, after looking at `Area`, he switched back to `Main` and `Height` roughly the same fraction of times. We can use the flow charts to compare subjects and easily see the differences in their transition probabilities. Subject 1's flow chart shows many narrow transitions between code locations, while Subject 2's flow chart (see report [4]) shows fewer, wider lines between consecutive code locations reflecting his more methodical comprehension style.

Second, we introduce VETtool[4] which can read a file of ELAN annotations (*i.e.*, the codes from the coding scheme) and display them on a timeline based on the duration of the associated fixations. After using ELAN to assign low-level codes (*e.g.*, `Signature`), VETool's visualization can show the areas of the program that were visited during the trial.

[4]VETool is GPLv2 software built on the NetBeans Platform with JavaFX. The source code can be found at `https://bitbucket.org/orlovpa/visual-evaluation-tool-vetool`.

Several codes can be presented together, allowing the analyst to see the timeline of the areas of interest (AOIs) overlaid with the patterns and strategies employed to understand them.

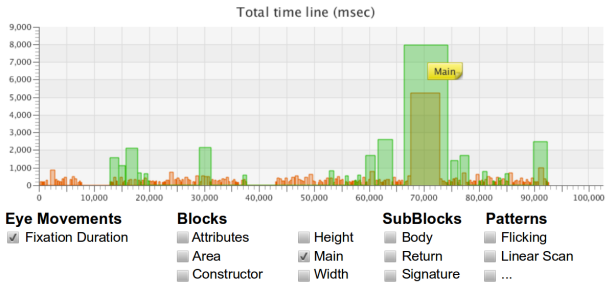

Figure 3: VETool displaying fixation durations (orange) and `Main` codes (green) for Subject 1

VETool also enables analysts to correlate fixation counts and durations with various low-level, pattern, or strategy codes, helping determine whether the subject was just briefly inspecting an area or examining it at length. Visualizing these correlations make it easier to compare two subjects by highlighting differences in their behavior. Figure 3 shows subject 1 concentrating on `Main` overlaid with fixation duration.

4.2 Quantizing Fixations

Next, we describe two static visualizations that we developed for understanding program comprehension. The first transforms fixations into coarse-grained areas of interest, such as single code elements, lines of code or blocks. The second plots the areas of interest (or metrics derived from them) on a timeline.

The workshop participants found coding the gaze location into code elements and blocks to be tedious and error-prone. Automatically coding these tiers would address a primary challenge of the eye tracking data analysis process [6]. So, we created a tool to draw virtual rectangles around areas of interest and assign each fixation to zero or more AOIs. Each AOI rectangle is then associated with the source code on the screen that represents each of the low-level codes in the coding scheme (*e.g.*, `Block`, `Line`, `Method Call`, etc.). For simplicity, assume that the AOI rectangles for the `Block`, `SubBlock`, `Signature`, and `Method Call` categories do not overlap, making them mutually exclusive (this is not the case for `Pattern`).

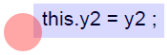

Figure 4: Sample assignment of a fixation to an AOI

To determine whether or not a fixation belongs to an AOI, the tool draws a circle around the center of the fixation point with radius R, and chooses the AOI rectangle with the largest area of overlap (Figure 4). The choice of the parameter R influences accuracy and depends on other parameters such as the size of the computer monitor and font size used in the experiment.

Once fixations from the eye gaze record have been assigned to AOIs (and thus the low-level codes in the coding scheme),

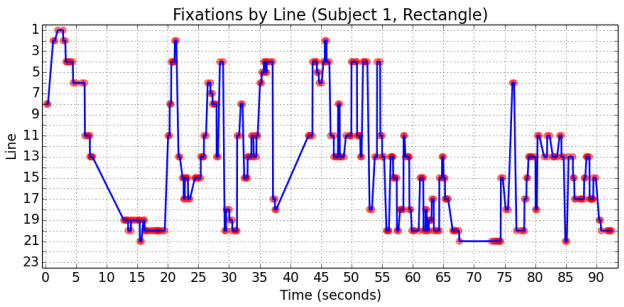

Figure 5: Fixations for Subject 1 quantized by line

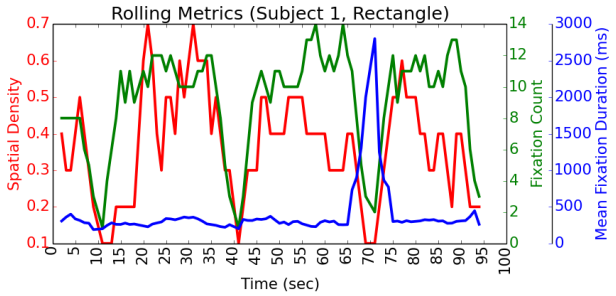

Figure 6: Fixation metrics for Subject 1

we plot them on a timeline. The timeline plot in Figure 5 shows Subject 1's fixations, quantized by `Line` using $R = 20$ pixels. These plots provide a wealth of information about a participant's behavior at a glance, enabling analysts to easily identify critical moments in the eye gaze record.

The AOI quantization tool is based on a graphical analysis of the eye gaze pixel locations on the screen. Its ability to track AOIs does not work so well on screens that contain scrolling or changing content. Applying it to IDEs in which the subject may scroll or type new code is simply infeasible for non-trivial programs and tasks [6]. Fortunately, there are IDE add-ons that can help, such as the iTrace plugin for Eclipse.[5] iTrace provides the exact source code entity a programmer looks at. Linking the eye tracker output directly to the IDE enables us to use the plugin to automatically annotate the eye gaze information with the correct program elements, solving the scrolling problem and the problem of adding or editing code.

Automatic code labeling facilitates aggregation of the data, which is valuable for group comparisons. In addition, if defined formally enough, pattern codes can be automatically derived from the low-level labels. For example, the `Linear Scan` pattern is readily apparently on a timeline plot. In Figure 5, we might say that `Linear Scan` describes the fixations between 2 and 20 seconds. We caution that someone should review the results because noise in the raw gaze data might have resulted in an incorrect classification of an AOI or low-level code, throwing off the pattern detector.

Finally, automatically labeling codes from our scheme offers a foundation for comparing results from future studies of eye tracking in computing education. We invite computing education researchers to apply our coding scheme in their own studies.

4.3 Strategies and Fixation Metrics

To aid in the identification and interpretation of Strategy codes, we compute three fixation metrics over the course of each trial: fixation count, mean fixation duration [19], and fixation spatial density [8]. We calculate each using a moving average of 4 second time windows, shifted by 1 second at a time. Typically, a single 4 second time window will contain about a dozen fixations.

The first metric is simply the total number of fixations in a time window. The second is the average duration of these fixations. The third metric divides the screen into a grid, and calculates the proportion of cells in the grid that contain at least one fixation. We applied a 10-cell, vertically-divided, rectangular grid to the source code editor, so a spatial density of 1 means that each rectangle in the grid was fixated upon at least once during the 4 second time window.

Finally, we plot the metrics (after removing time windows which contain no fixations) on a timeline. Figure 6 illustrates the three metrics computed from Subject 1's trial. Dips in the spatial density (shown on the red line) correspond to time windows in which Subject 1 focused on just one or two lines of code. Sometimes this corresponds with an increase of the number of fixations (blue line), we found to be useful to distinguish between `Debugging` and `Test Hypothesis` Strategy codes.

Subject 1's erratic gaze is revealed by the very low mean fixation duration (green line), however, we see it increase sharply just after 70 seconds into the trial when Subject 1 focuses his gaze on the final line of the program: `System.out.println(rect2.area());` Recall that his task in this trial was to obtain the value of `rect2.area()`. Given the increased mean fixation duration, and the drop in fixation count and spatial density at approximately 65-75 seconds, we hypothesize that Subject 1 is performing the necessary mental calculation to compute the area of `rect2`. While it may not be possible to pinpoint changes in strategy using this kind of visualization, we can quickly identify interesting time windows that we should investigate further.

5. PROSPECTS FOR PROGRAMMING EDUCATION

Eye tracking offers opportunities for a great range of research questions in the areas of programming education and program comprehension. Possible research topics include

- the effects of text-based, graphical, or UML program representations [26]; syntax and language features; programming paradigms;
- the behaviors and strategies of a learner's reading, understanding, writing and debugging tasks'
- challenges for learners, *e.g.*, what makes tasks difficult for them, what obstacles impair their understanding and use of programming concepts;
- evaluation of tools for static and dynamic program visualization [5], as well as for IDEs [3]; and
- gaze-related concerns, *e.g.*, exploring the possibility of providing immediate feedback based on eye movements in programming environments [2].

[5] `http://www.csis.ysu.edu/~bsharif/iTrace`

These topics can lead to advances in teaching programming in the following areas.

5.1 Decoding the Learner

Eye tracking allows us to retrace how the novice goes about reading and understanding source code. We can obtain information about difficulties, behavior and strategies, and develop new tools to assess learners. In addition, we can describe and evaluate an individual's level of expertise on the basis of aggregated empirical studies of eye tracking that compare novice and expert programmers.

5.2 Advances in Teaching Material and Tools

The detailed data provided by eye trackers can also help to advance learning tools, such as IDEs for learning programming. Eye tracking is a well-established instrument in usability testing with a large corpus of analyses, metrics and examples of best practice. Applying eye tracking can help developers increase tool usability and lower the barriers to adoption. More sophisticated uses of eye tracking could provide highly-contextualized feedback to the learner, for example in an automated tutor. If an eye tracking metric moves past a particular threshold, it could indicate that the student is having difficulties with the material, and could use a hint in order to make progress. Visual cueing could be employed in an IDE, if students look too long at the wrong section of code, or thrash their gaze over the entire program without focusing on any particular part.

5.3 New Perspectives for Teaching Code Reading

Eye tracking studies can be used to shed light on how individuals conceptualize and discern the embedded process of computational representations. A crucial challenge yet to be solved is to explore the code reading characteristics of individuals. First a normative or generalized pattern needs to be established, *e.g.*, a program flow gaze pattern for specific source code examples. After taking into account the learner's ability or level of understanding or the difficulty of the task, individual deviations from this normative eye gaze pattern can be used to reveal meaningful information about the concrete learning process.

These normative eye gaze patterns can help identify differences in the strategies adopted by programmers with various levels of expertise, domain knowledge, and skills. Given that the ability to read and comprehend code seems to be linked with the ability to write it, there is substantial evidence that many programming novices have not yet acquired the ability to effectively read programs [16]. As with natural language, concrete code reading skills can explicitly be taught to learners, addressing purposes like debugging code or working out why it is written that way. It is hard to do this now because reading and debugging strategies are so ingrained that people are not aware of them on a conscious level.

Eye tracking could be used to raise a novice's awareness of how they go about reading code. A novice could track himself solving a task, and later understand how what he was thinking corresponded with where he was looking. This would build a novice's self-awareness and facilitate the metacognition that eventually teaches a beginner to read code efficiently.

We can use eye gaze data to explore experts' strategies and develop teaching materials demonstrating their application. Novices observing expert programmers' eye movements on a given task could get visual cues as to what is important [22]. This might lead to the creation of a tool for teaching reading skills that shows the student where to look, an approach that has proved successful in other domains [25]. If students could be taught to consciously use code reading strategies according to code size, program structure and other parameters, they could see how to make their own approach more effective. After they have been taught these reading techniques, teachers can use eye tracking to verify that students are using them, enabling better assessment of teaching interventions.

From our workshop examples, we saw that experts can read the same code using very different strategies. Therefore, the ones offered to learners should be those that were used consistently by many expert programmers. Individual students could then adopt a strategy that fits well with their own personal approach.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented eye tracking as a method to enrich computing education research. Tracking a person's gaze gives a record of their visual behavior on a super fine-grained scale in both space and time. We can build on the small, but growing, body of sound work on eye tracking in the context of programming to enhance programming education practice and research. Eye tracking has the potential to open up new methods of understanding how people program and learn to program, of corroborating existing empirical research, and of tackling currently unsolved questions. Its benefits will stretch across broad areas of research, such as program reading and comprehension, and afford new teaching material for aspects of professional expertise that have not yet been analyzed.

Gaze analysis offers intriguing prospects for further study. Eye tracking can record a person's visual behavior during reading, without interruption, adding to his cognitive load, or requiring a subjective report. Thus, this research tool provides a new quality and directness, along with a much finer data granularity, to observe cognitive processing.

Gaze analysis can also serve as an additional source of data to corroborate studies carried out with other research methods. Such study replications will help to increase our collective evidence and sharpen our theories. With more and more of the challenges associated with eye tracking being solved, with eye trackers becoming more affordable, and with relevant analytical tools becoming increasingly available, such studies might even become a standard in educational research.

Our Eye Movements in Programming workshop and this paper are a first step towards making eye tracking more accessible to computing educators. When a dozen experts in computer science education and eye tracking can agree on the potential of using eye movement data in programming education, their position must surely have some merit.

7. ACKNOWLEDGMENTS

We would like to thank all workshop participants for their great work.

8. REFERENCES

[1] A. Allevato and S. H. Edwards. Discovering patterns in student activity on programming assignments. In *ASEE Southeastern Section Annual Conference and Meeting*, 2010.

[2] V. M. G. Barrios, C. Gütl, A. M. Preis, K. Andrews, M. Pivec, F. Mödritscher, and C. Trummer. Adele: A framework for adaptive e-learning through eye tracking. In *Proc. of IKnow*, volume 4, pages 1–8. Citeseer, 2004.

[3] R. Bednarik. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International J. of Human-Computer Studies*, 70(2):143–155, 2012.

[4] R. Bednarik, T. Busjahn, and C. Schulte. Eye movements in programming education: Analyzing the expert's gaze. Technical report, University of Eastern Finland, Joensuu, Finland, 2014.

[5] R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen. Effects of experience on gaze behavior during program animation. In *Proc. of 17th Annual Workshop of the Psychology of Programming Interest Group*, pages 49–61, Sussex University, 2005.

[6] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proc. of the Symposium on Eye Tracking Research & Applications*, pages 125–132. ACM, 2006.

[7] T. Busjahn, C. Schulte, and A. Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proc. of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9, Koli, Finland, 2011. ACM.

[8] L. Cowen, L. J. Ball, and J. Delin. An eye movement analysis of web page usability. In *People and Computers XVI-Memorable Yet Invisible*, pages 317–335. Springer, 2002.

[9] M. E. Crosby and J. Stelovsky. How do we read algorithms? A case study. *Computer*, 23(1):24–35, 1990.

[10] Q. Fan. *The effects of beacons, comments, and tasks on program comprehension process in software maintenance*. PhD thesis, University of Maryland at Baltimore County, Catonsville, MD, USA, 2010.

[11] J. Helminen, P. Ihantola, and V. Karavirta. Recording and analyzing in-browser programming sessions. In *Proc. of the 13th Koli Calling International Conference on Computing Education Research*, pages 13–22, Koli, Finland, 2013. ACM.

[12] V. A. Huemer, M. Hayashi, F. Renema, S. Elkins, J. W. McCandless, and R. S. McCann. Characterizing scan patterns in a spacecraft cockpit simulator: Expert vs. novice performance. *Proc. of the Human Factors and Ergonomics Society Annual Meeting*, 49(1):83–87, Sept. 2005.

[13] A. Hyrskykari, S. Ovaska, P. Majaranta, K.-J. Räihä, and M. Lehtinen. Gaze path stimulation in retrospective think-aloud. *J. of Eye Movement Research*, 2(4):1–18, 2008.

[14] M. Just and P. Carpenter. A theory of reading: From eye fixations to comprehension. *Psychological Review*, 87:329–354, 1980.

[15] M.-J. Laakso and Simon, editors. *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, November 2013.

[16] R. Lister, C. Fidge, and D. Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *SIGCSE Bulletin*, 41(3):161–165, 2009.

[17] J. C. Nordbotten and M. E. Crosby. The effect of graphic style on data model interpretation. *Information Systems J.*, 9(2):139–155, 1999.

[18] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proc. of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, NY, USA, 2012. ACM.

[19] A. Poole and L. J. Ball. Eye tracking in human-computer interaction and usability research: Current status and future. In *Prospects, Chapter in C. Ghaoui (Ed.): Encyclopedia of Human-Computer Interaction. Pennsylvania: Idea Group, Inc*, 2005.

[20] B. Sharif, M. Falcone, and J. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proc. of the Symposium on Eye Tracking Research & Applications*, pages 381–384, Santa Barbara, CA, 2012. ACM.

[21] S. Sridharan, R. Bailey, A. McNamara, and C. Grimm. Subtle gaze manipulation for improved mammography training. In *Proc. of the Symposium on Eye Tracking Research & Applications*, pages 75–82, Santa Barbara, California, 2012. ACM.

[22] R. Stein and S. E. Brennan. Another person's eye gaze as a cue in solving programming problems. In *Proc. of the 6th international conference on Multimodal interfaces*, pages 9–15, PA, USA, 2004. ACM.

[23] R. Turner, M. Falcone, B. Sharif, and A. Lazar. An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proc. of the Symposium on Eye Tracking Research & Applications*, pages 231–234, Safety Harbor, Florida, 2014. ACM.

[24] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proc. of the Symposium on Eye Tracking Research & Applications*, pages 133–140, San Diego, California, 2006. ACM.

[25] S. J. Vine, R. S. Masters, J. S. McGrath, E. Bright, and M. R. Wilson. Cheating experience: Guiding novices to adopt the gaze strategies of experts expedites the learning of technical laparoscopic skills. *Surgery*, 152(1):32–40, July 2012.

[26] S. Yusuf, H. Kagdi, and J. I. Maletic. Assessing the comprehension of UML class diagrams via eye tracking. In *Proc. of the 15th IEEE International Conference on Program Comprehension*, pages 113–122, 2007.