

# The impact of syntax colouring on program comprehension

Advait Sarkar

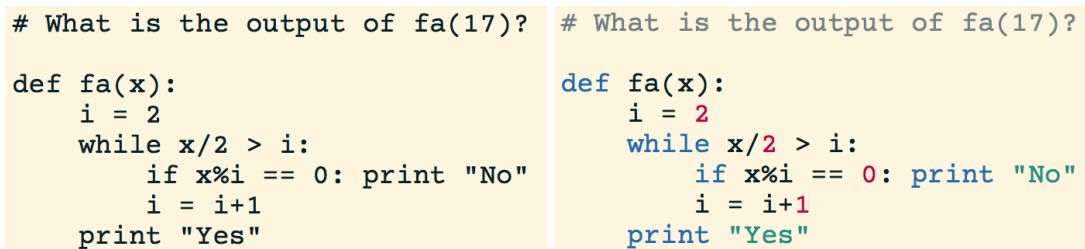
Computer Laboratory, University of Cambridge  
Cambridge, UK  
advait.sarkar@cl.cam.ac.uk

**Abstract.** We present an empirical study investigating the effect of syntax highlighting on program comprehension and its interaction with programming experience. Quantitative data was captured from 10 human subjects using an eye tracker during a controlled, randomised, within-subjects study. We observe that syntax highlighting significantly improves task completion time, and that this effect becomes weaker with an increase in programming experience.

Keywords: POP-II.B. Program Comprehension; POP-III.D. Editors; POP-V.B. Eye-tracking

## 1 Introduction

Syntax colouring, commonly known as *syntax highlighting*, is a feature of some text editors which colours lexical tokens in source code text according to a certain categorisation. An example of how source code may look with and without syntax highlighting can be found in Figure 1.



<pre># What is the output of fa(17)?  def fa(x):     i = 2     while x/2 &gt; i:         if x%i == 0: print "No"         i = i+1     print "Yes"</pre>	<pre># What is the output of fa(17)?  def fa(x):     i = 2     while x/2 &gt; i:         if x%i == 0: print "No"         i = i+1     print "Yes"</pre>
--	--

Fig. 1. Left: code without highlighting. Right: same code with syntax highlighting.

Syntax highlighting is included in text editors on the premise that it makes working with code easier. It is easier to spot certain kinds of syntactic bug with syntax colouring: for instance, if string literals are assigned a unique colour, then an unclosed string is easy to spot as all subsequent text until the next delimiter is coloured. It is important to note that syntax highlighting is purely secondary notation (T. R. Green, 1989) meant for humans; it does not affect the behaviour of the code.

Close attention has not yet been paid to whether syntax highlighting has any impact on the speed, precision or ease with which a programmer can internalise a program’s semantics, i.e. *comprehend* the program. A plausible intuition is that the improved readability and clearer structure of syntax-highlighted code would make it “easier” to understand the behaviour of a program, but the nature and degree of any such improvement has not previously been investigated.

In this study we establish whether source code syntax colouring has an effect on program comprehension time, and furthermore investigate whether this effect varies by programming experience. It is conceivable that a difference in comprehension speed might be attributable to differences in eye fixation patterns on syntax-highlighted keywords, on the basis that visual attention (focus on a particular location) triggers mental processes to comprehend or solve a given task (Just & Carpenter, 1980). We conduct an eye-tracking study, explained in §2, to substantiate this. These research objectives are formalised as hypotheses in §3. Finally, we discuss our findings in §4.

## 1.1 Previous work

Most previous work on colour in text has focused on English prose and not source code. Evidence suggests that colouring tokens in text has no effect on the speed of visual search when the colour of the target is unknown, but when the colour of the target is known, search times are considerably smaller (B. F. Green & Anderson, 1956; Rubin, 1988). Colour has been found to be useful for emphasising format and categorisation (Van Nes, 1986; Rubin, 1988). By contrast to the corresponding research in print media, Hakala et al. found that syntax highlighting did not have a significant impact on the speed of visual search (2006) onscreen.

Baecker (1988) studied aspects of typography with the specific purpose of enhancing source code readability and comprehensibility. The author prepared short programs, varying several aspects of the layout and typography, such as punctuation marks, kerning, and incorporation of colour. The prepared program along with a questionnaire was presented to the participants of the study. The impact of the improved layout on readability, comprehensibility, and recall was measured as a function of the number of questions answered correctly and the time taken to answer the questions. It was found that the improved layout increased the mean number of correct answers by 11%, and that program readability was improved by 25%.

Bednarik and Tukiainen (2006) demonstrated the feasibility of using eye-tracking data to study program comprehension, later using their methods to study the evolution of debugging strategies (Bednarik & Tukiainen, 2008). Sharif and Maletic (2010) used eye tracking to study the impact of identifier naming conventions (e.g. “camelCase” and “under\_score”) on visual search. Participants visually searched for a known identifier amongst a word cloud of similar identifiers. Fixation counts and durations were used to measure “visual effort.” The underscore style was found to require lower matching times and visual effort than the camel case style. A summary of other work on eye-tracking in programming is given by Busjahn et al. (2014).

Gilmore and Green (1988) conducted a study wherein bugs were introduced into program fragments containing simple loop and conditional structures. Some bugs were surface level, like misspellings, and some were at a deeper level, like putting an assignment in the wrong place. A highlighter was used in combination with indentation to highlight control structures, cognitive-plan structures, both, or neither (a ‘cognitive plan’ relates to the programmer’s mental model for the algorithmic structure of the program (Spohrer, Soloway, & Pope, 1985)). The participants were to find the bugs. Participants found surface bugs faster with control-structure cues, and deep bugs faster with cognitive-plan colouring. The authors conclude, importantly, that the added information is only effective if it relates to the task. Although the colouring was not syntax-based, this relates to our final conclusion that highlighting improves comprehension speed.

## 2 Experimental methodology

### 2.1 Comprehension task design

The comprehension task took the form of a *mental execution*, where the participants were given a function definition and requested to compute its output for a given set of arguments.

An example task is presented below:

```
What is the output of da([1,2],[5,6],3)?
def da(list1, list2, x):
    list3 = []
    for j in list1:
        for k in list2:
            list3.append(j+k)
    for e in list3:
        if (e%x) == 0: print e
```

To facilitate a within-subject comparison, each participant performed the task with *pairs* of programs, where one of the pair had syntax highlighting, and the other did not. Tasks were carefully designed to have a difficulty comparable to their paired counterparts by choosing programs that were of identical structure but with certain specific variations that had a major impact on their semantics. These variations included reversing the direction of an inequality, replacing additions with subtractions, or interchanging the arguments. Thus, tasks in a pair had comparable difficulty but very little transferable knowledge, effectively minimising order effects.

The participants were given 3 pairs of comprehension tasks. From each pair, the task to be highlighted was randomly chosen to reduce the likelihood of highlighting and difficulty becoming artificially correlated. We ensured ample difference between each pair of tasks, and randomised the order of tasks for each participant to account for repeated testing and order effects.

Since we intended to investigate the impact of programming experience, we chose to present the tasks in a language in which the available participants, graduate computer science students, had varying levels of experience. Python was the appropriate choice for two reasons:

1. The graduate students had a broad range of undergraduate backgrounds, so much variation in Python experience could be expected: some participants had been taught extensive courses, whereas some had never encountered Python prior to the experiment.
2. It is similar to other popular languages such as Java, C++, and even pseudocode, so a specific lack of Python experience would not make the task impossible to complete.

Python differs in two ways from languages with which the participants are more likely to be familiar: its use of indentation to indicate block structure, and its lack of explicit variable types. Participants were made aware of both of these issues before they began the task. To avoid datatype-related confusion, a uniform variable naming scheme was adopted in the tasks. For example, integers were named `x`, `y`, etc. and lists were named `list1`, `list2`, etc.

A final consideration is the colouring scheme of the syntax highlighting. It was not within the scope of this study to investigate the impact of different colouring schemes, although it would make for interesting future work. We focus on token-level colouring, as opposed to block-level colouring such as that proposed by Cigas (1990), since token-level colouring is far more prevalent in practice. We used the Solarized<sup>1</sup> palette, which has been designed for usability.

## 2.2 Eye-tracking apparatus

We used a Tobii<sup>2</sup> X120 eye tracker, a video-based remote eye tracker that captures eye movements using twin infrared cameras. No head gear is necessary. The tracker is placed on the desk directly underneath the monitor. For optimal tracking, the tracker is configured to be aware of its location with respect to the screen and of the screen's dimensions and resolution. Sampling occurs at a temporal resolution of 60Hz with a latency of 25-35ms and accuracy of the order of 0.5°. The eye tracker compensates for head movement to some extent and is able to recover if the subject temporarily looks away from the screen. The eye gaze data includes timestamps of fixations, onscreen fixation coordinates, fixation durations, eye positions, pupil size, etc.

## 2.3 Measuring programming experience

The participants were asked to complete a questionnaire recording the following:

1. How long the participant has been programming.
2. A self-reported score from 1-10, 10 being "highly experienced".

<sup>1</sup> Solarized - Ethan Schoonover, <http://ethanschoonover.com/solarized> (as of June 15, 2015)

<sup>2</sup> Tobii Eye-Tracking Research, <http://www.tobii.com/en/eye-tracking-research/> (as of June 15, 2015)

3. The participant’s self-estimated peer group decile (e.g. top 10%, top 20%, etc.)
4. For each language the participant has used (up to a maximum of 4):
  - How long the participant has been using that language.
  - How often the participant uses it.
  - The largest or most complex program the participant has written in it.
  - A self-reported proficiency score from 1-10, 10 being highly proficient.

Since programming experience is complex and multifaceted, it is impossible to measure along an objective, interval scale. Thus, these records were used purely to establish an ordinal ranking of participants by experience. The answers to questions 1-3 were used to perform a pairwise comparison of participants’ experience, and the answers to the question 4 provided enough information to adjust any unrealistic answers<sup>3</sup> provided for the first 3 questions.

## 2.4 Experimental procedure

The experimental procedure was explained to the participant, including a brief explanation of the block structure and type inference in Python. The participant was seated approximately 70cm in front of the screen and eye tracker. The participant read and signed an informed consent form. Next, the eye tracker was calibrated using a nine-point calibration lasting approximately 45 seconds, during which the participant focused their eyes on nine points that appear on an equally spaced 3×3 onscreen grid in random order.

To begin with, the participant completed two example tasks, one with highlighting and one without. No measurements from these tasks were included in subsequent analysis. This allowed the participant to become familiar with the nature of the task and also ask questions if necessary. Once the participant understood the task, the actual study commenced. The participant stepped through each of 3 task pairs, 6 tasks in all, in random order. The tasks were presented in a slideshow for instant transitions between tasks. Timing and eye-tracking for each task began when the slideshow switched to the task slide, and ended when the participant identified the correct answer by saying the result of the requested computation aloud. Participants were requested to complete the tasks mentally without the aid of pen and paper, but were allowed to vocalise their thinking process. The experiment had an average duration of 13 minutes, never exceeding 20. Finally, participants completed the programming experience self-assessment questionnaire.

We gathered data from 10 graduate computer science students at the University of Cambridge. For each participant we recorded an average of ~1300 fixation events. The eye-tracker recorded particularly poor fixation data for 3 participants; this was attributable to the fact that these participants wore glasses which confused the eye tracker. Their data was excluded from analyses of visual effort.

## 3 Results

In this section we report the results of the study previously outlined. The primary variables being studied are presented in Table 1. In §3.2 we introduce two derived variables which are not included in this table. We use the Shapiro-Wilk test to establish normality. We use the Wilcoxon signed rank test (WSRT) for paired nonparametric comparisons.

---

<sup>3</sup> We made the anecdotal observation that experienced programmers tended to underestimate their competency, whereas less experienced programmers tended to gauge their ability more accurately. Evidence, perhaps, that the Dunning-Kruger effect applies to programming experience (Kruger & Dunning, 1999).

**Table 1.** Variables

<b>Highlighting</b>	Whether the task was highlighted or plain. An independent variable.
<b>Experience</b>	Each of our 10 participants was assigned a rank from 1-10 with 10 being most experienced. An independent, ordinal variable.
<b>Task completion time</b>	The time it takes a participant to complete a task (in seconds).
<b>Fixation count</b>	The number of times a participant fixates on the task image while completing it. A dependent, interval variable.
<b>Fixation duration</b>	How long an individual fixation lasts (in microseconds).
<b>Prompt fixations</b>	The task image was divided into two so-called “areas of interest”: the <i>prompt</i> , which was the single line containing the instruction for the task, and the <i>content</i> , which was the code for the task. This variable counts a participant’s fixations on the prompt during a task.
<b>Context switches</b>	The number of times a participant fixates on an area of interest that is different from the area that was the subject of the immediately previous fixation. For example, if the participant fixates on the prompt, then on the content, and then on the prompt again, the number of switches is 2.

### 3.1 Effects of highlighting

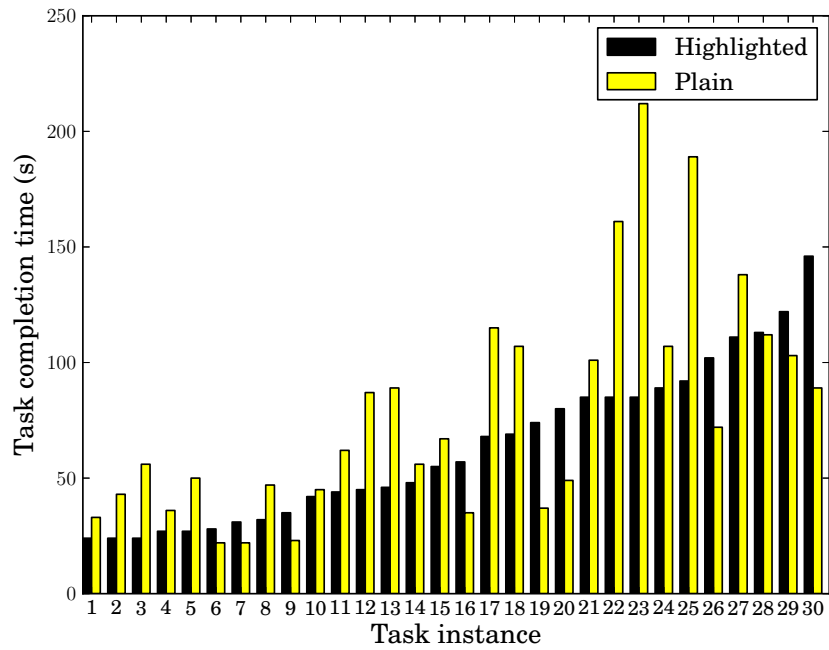
The task completion times for highlighted as well as plain tasks were not normally distributed. Task completion times for highlighted versions of the tasks were significantly lower (WSRT:  $T = 136, p = 0.047$ ). The difference in medians is 8.4s. This is illustrated in Fig. 2.

We investigated the effect of highlighting on some features of the eye-tracking data, namely fixation durations, fixation count, fixations on prompt and context switches. These can be collectively thought of as the “visual effort” required to perform the comprehension task. We did not find a significant effect of highlighting on fixation counts, fixation durations or prompt fixations.

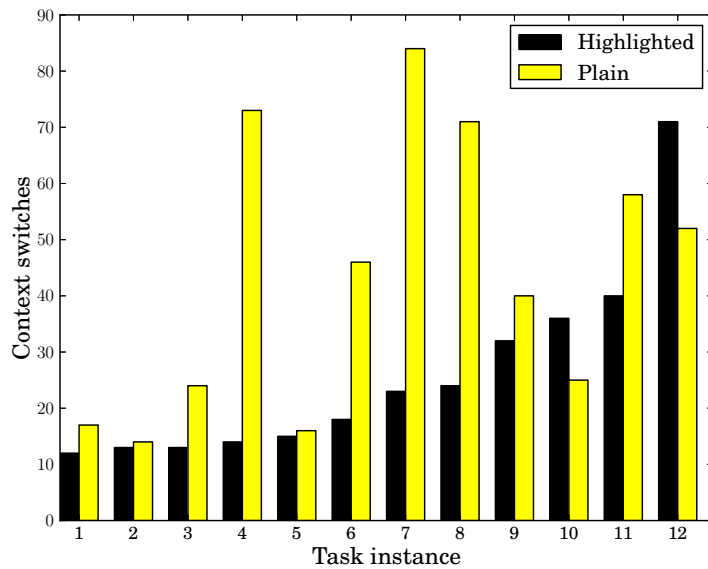
Recall that a context switch occurs when a participant fixates on an area of interest that is different from the area that was the subject of the immediately previous fixation. Highlighted code incurs significantly fewer context switches than non-highlighted code (WSRT:  $T = 13.5, p = 0.045$ ). The difference in medians is 23 context switches. This is illustrated in Fig. 3.

### 3.2 Effect of programming experience

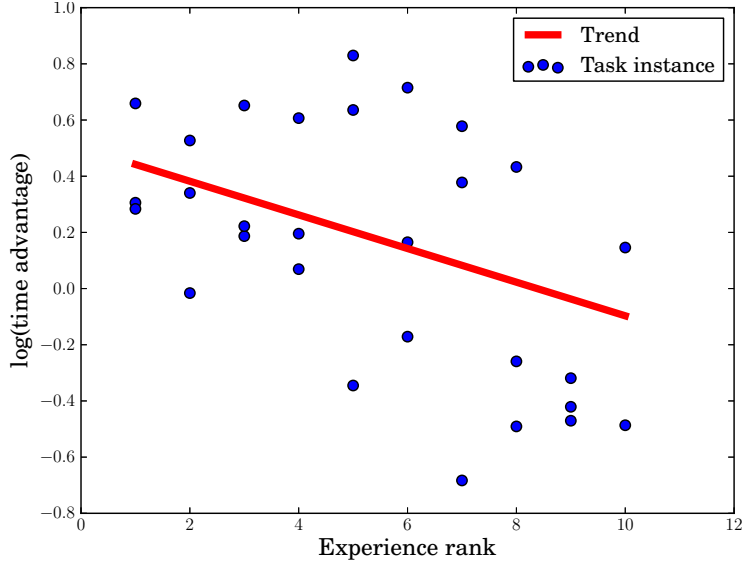
As the data was not normally distributed, a 2-way ANOVA could not be used to investigate the interaction of experience with highlighting on task times. We instead introduce a new derived variable, *time advantage*, which is simply the ratio of the task completion time in the non-highlighted task to the completion time for its highlighted counterpart. Thus, if a participant completed the plain version of a task in 60s, and the highlighted counterpart in 30s, the time advantage for that task instance is  $60s/30s = 2$ . We consider this to embody the effect of highlighting on task completion time. We then investigated the correlation of experience with this variable. On the raw data we observe a significant Spearman correlation of -0.37 ( $p = 0.044$ ) but a weaker Pearson correlation ( $r = -0.29, p = 0.12$ ), suggesting a nonlinear relationship. When log-normalised, the time advantage variable has a significant Pearson correlation with experience ( $r = -0.39, p = 0.033$ ). We conducted a similar study with context switches for the derived variable *switch advantage*, but did not find a significant correlation.



**Fig. 2.** Bar graph comparing task completion times for a highlighted task with its plain counterpart. Each pair of bars is an instance of a participant completing a particular task pair.



**Fig. 3.** Bar graph comparing context switches for a highlighted task with its plain counterpart. Each pair of bars is an instance of a participant completing a particular task pair. Fewer instances were available for this comparison since data from participants wearing glasses had to be excluded from analyses of visual effort.



**Fig. 4.** Each point represents a single task pair completed by a single participant. Its  $x$ -coordinate is the rank of the participant in ascending order of experience, and the  $y$ -coordinate is the logarithm of time advantage. The trend line is given by least-squares regression, has the slope  $-0.06$ , and estimated parameter covariance  $< 10^{-3}$ .

## 4 Discussion

Our results confirm the intuitive but heretofore unsubstantiated idea that syntax highlighting in source code improves program comprehension. The median difference in task completion time was 8.4s in favour of highlighting. It is possible that the magnitude of this effect increases with the length of the program; an interesting area for future investigation.

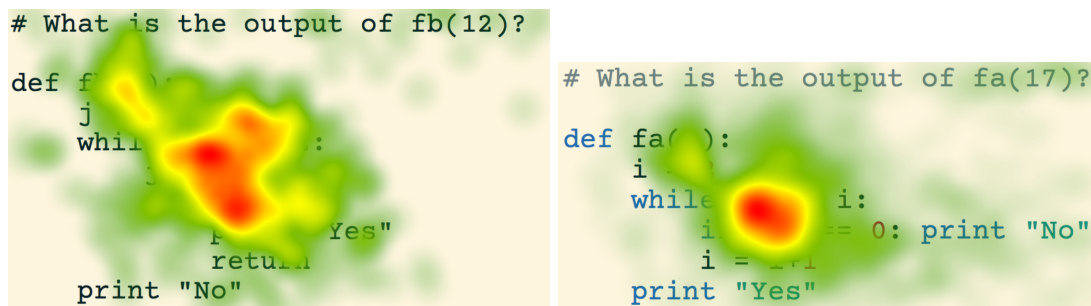
We were unable to detect differences in fixation count and durations due to highlighting. Since 3 of our 10 participants were excluded from analyses of visual effort as their glasses rendered the eye-tracking data unusable, it is possible that statistical significance was not achieved as a consequence of the reduced sample size. It is also plausible that syntax highlighting genuinely does not affect fixation durations or count. As shown by the observation that syntax highlighting has an effect on context switches, the fixation sequence might be altered in a far more sophisticated manner.

Our inference that the presence of highlighting significantly reduces the number of context switches is of particular interest. Context switches during the experimental tasks almost always took the same form: while reading the content, the participant glanced at the prompt before continuing with the content. This sequence of 2 switches always served a single purpose: to remind the participant of the argument values. Our results show that the need to be reminded of the input values was significantly greater when the code was not highlighted.

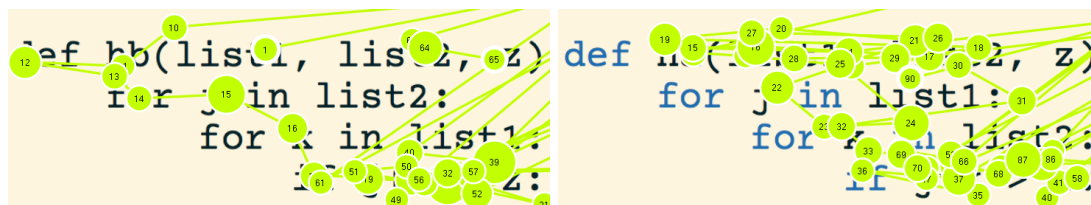
There is no immediately clear explanation for this, but we suggest a simple one here: it is plausible that the mental overhead required to process and understand plain code is greater than the mental overhead required to process highlighted code, since highlighted code contains additional semantic richness by virtue of the colours of the tokens. This additional overhead in plain code causes other items, such as the values of the input arguments, to be displaced from the working memory (Baddeley & Hitch, 1974) of the participant. This would account for the increase in context switches in non-highlighted code.

The theory that there is less overhead involved in processing code with additional visual cues (such as those provided by syntax colouring) is qualitatively supported by our results, as well as quantitatively by the aforementioned study by Gilmore and Green (1988). The data from some participants suggests that syntax highlighting allows the programmer to focus on a

smaller region of code, as illustrated in the fixation heat map in Figure 5. Furthermore, while the quality of the eye-tracking data was not generally good enough to designate single keywords as areas of interest, the data from certain participants with very good readings suggests that syntax highlighting allows the programmer to even ignore some keywords entirely, as illustrated in the gaze plot in Figure 6. This theory is also supported by our inability to detect a relationship between experience and the effect of highlighting on context switches. If highlighting does affect context switches by the mechanism hypothesised above, there is no reason to expect it to vary by programming experience. Future work may study this effect in greater detail, providing better quantitative evidence.



**Fig. 5.** Left: fixation heat map for code without highlighting. Right: heat map for the highlighted counterpart.



**Fig. 6.** Left: gaze plot for code without highlighting. Right: gaze plot for the same participant with the highlighted counterpart. The numbers denote the order in which the fixations occurred. Observe the lack of fixations on keywords in the highlighted case.

Finally, we found that programming experience was negatively correlated with time advantage. It appears that syntax highlighting improves program comprehension speed to a greater extent in novice programmers than in experienced programmers. However, this may be a consequence of the brevity of the tasks; repeating the study with longer programs may reveal that experienced programmers stand to gain just as much as less experienced programmers.

## 5 Conclusions

We investigated the effect of syntax highlighting on program comprehension and its interaction with programming experience using eye-tracking data captured from 10 participants. Each participant was requested to mentally compute the output of a Python function for a given set of arguments, repeating this several times with highlighted and non-highlighted code.

The presence of syntax highlighting significantly reduces task completion time, but the magnitude of this effect decreases as programming experience increases. We did not detect an effect of the presence of highlighting on the durations of individual fixations, nor did we detect an effect on the total number of fixations required to complete the task.



The presence of syntax highlighting significantly reduces context switches. We hypothesise that syntax highlighting improves the ability of the programmer to mentally retain the state of the execution, and that highlighted code incurs a lower mental comprehension overhead. In some cases, the eye-tracking data suggested that the participants were able to ignore highlighted keywords entirely, as though perceiving them peripherally was enough to incorporate their semantics into the computation. Future work may investigate this effect, as well as the effect on longer programs, with more quantitative rigour.

## 6 Acknowledgements

Many thanks to Graham Titmus for help with hardware; the Rainbow research group<sup>4</sup> for providing the eye tracker and laboratory space; Alan Blackwell for various discussions on the topic; Thomas Green for his considerate, detailed and excellent feedback on the written drafts; and all the participants for their valuable time and effort.

## References

- Baddeley, A. D., & Hitch, G. J. (1974). Working memory. *The psychology of learning and motivation*, 8, 47–89.
- Baecker, R. (1988). Enhancing program readability and comprehensibility with tools for program visualization. *Proc. 10th International Conference on Software Engineering, 1988*, 356–366.
- Bednarik, R., & Tukiainen, M. (2006). An eye-tracking methodology for characterizing program comprehension processes. *Proceedings of the 2006 symposium on Eye tracking research & applications (ETRA 2006)*, 125–132.
- Bednarik, R., & Tukiainen, M. (2008). Temporal eye-tracking data: evolution of debugging strategies with multiple representations. *Proc. ETRA 2008*, 99–102.
- Busjahn, T., Schulte, C., Sharif, B., Begel, A., Hansen, M., Bednarik, R., ... others (2014). Eye tracking in computing education. In *Proceedings of the tenth annual conference on international computing education research* (pp. 3–10).
- Cigas, J. F. (1990). Dynamically displaying a pascal program in color. *Proceedings of the 1990 ACM SIGSMALL/PC symposium on Small systems*, 68–71.
- Gilmore, D., & Green, T. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology*, 40(3), 423–442.
- Green, B. F., & Anderson, L. K. (1956). Color coding in a visual search task. *Journal of Experimental Psychology*, 51(1), 19.
- Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.
- Hakala, T., Nykyri, P., & Sajaniemi, J. (2006). An experiment on the effects of program code highlighting on visual search for local patterns. *Psychology of Programming Interest Group*, 38–52.
- Just, M. A., & Carpenter, P. A. (1980). A theory of reading: From eye fixations to comprehension. *Psychological review*, 87, 329–354.
- Kruger, J., & Dunning, D. (1999). Unskilled and unaware of it: how difficulties in recognizing one’s own incompetence lead to inflated self-assessments. *Journal of personality and social psychology*, 77(6), 1121.
- Rubin, T. (1988). *User interface design for computer systems*. Halsted Press.
- Sharif, B., & Maletic, J. I. (2010). An eye tracking study on camelcase and under\_score identifier styles. *18th International Conference on Program Comprehension (ICPC 2010)*, 196–205.

<sup>4</sup> Computer Laboratory: Graphics & Interaction Group, <http://www.cl.cam.ac.uk/research/rainbow/> (as of June 15, 2015)

- Spohrer, J. C., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy pascal programs. *Human-Computer Interaction*, 1(2), 163–207.
- Van Nes, F. (1986). Space, colour and typography on visual display terminals. *Behaviour & Information Technology*, 5(2), 99–118.