Booth, T. & Stumpf, S. (2013). End-user experiences of visual and textual programming environments for Arduino. Lecture Notes in Computer Science, 7897 L, pp. 25-39. doi: 10.1007/978-3-642-38706-7_4

End-User Experiences of Visual and Textual Programming Environments for Arduino

Tracey Booth and Simone Stumpf

City University London

{tracey.booth.2, simone.stumpf.1}@city.ac.uk

Abstract. Arduino is an open source electronics platform aimed at hobbyists, artists, and other people who want to make things but do not necessarily have a background in electronics or programming. We report the results of an exploratory empirical study that investigated the potential for a visual programming environment to provide benefits with respect to efficacy and user experience to end-user programmers of Arduino as an alternative to traditional text-based coding. We also investigated learning barriers that participants encountered in order to inform future programming environment design. Our study provides a first step in exploring end-user programming environments for open source electronics platforms.

Keywords: End-user programmers, Arduino, Visual Programming

1 Introduction

Open source hardware platforms such as Arduino [23] and Raspberry Pi [32] have reinvigorated interest in hacking and tinkering to create interactive electronics-based projects. These platforms present an opportunity for end users to move beyond being mere consumers of technology to being producers of it. Arduino is based on a simple microcontroller board (Figure 1) that was designed for use in personal projects, even by people with little electronics or programming experience – artists, hobbyists, or children.

However, although easier to learn than many other programming languages, the Arduino programming language still requires some skill to master and for potential end-user programmers of Arduino this means getting to grips with coding. Recently, visual programming languages (VPLs) for Arduino have been developed and we wondered whether a visual representation might provide an easier route of entry into programming. This fits with the rise of other visual programming environments aimed, for example, at children to engage them and facilitate the learning process. Would a VPL have similar benefits for adult users beginning to learn to program an electronics platform? Similarly, some work has started to emerge to investigate learning barriers for specific end-user programming environments [5, 11, 12]. Conse-

quently, we wondered what barriers end users encounter when programming electronics platforms.

In this paper we describe and discuss findings from an empirical study with adult novice end-user programmers of Arduino, using textual and visual programming environments. The overall aim of our research was to investigate whether a visual programming language for Arduino offers benefits over a traditional, text-based language for adult end-user programmers, specifically in terms of efficacy and user experience. We were also interested in what barriers end users currently face with a view to improving these environments.

Our study is the first of its kind (as far as we know) to investigate a VPL for Arduino. We contribute novel insights into understanding how end users interact with these programming environments in the growing area involving physical prototyping and study their benefits, costs and learning barriers. Our findings can provide the basis for improving the design of visual programming environments for the Arduino, and potentially also programming environments suitable for other related domains.

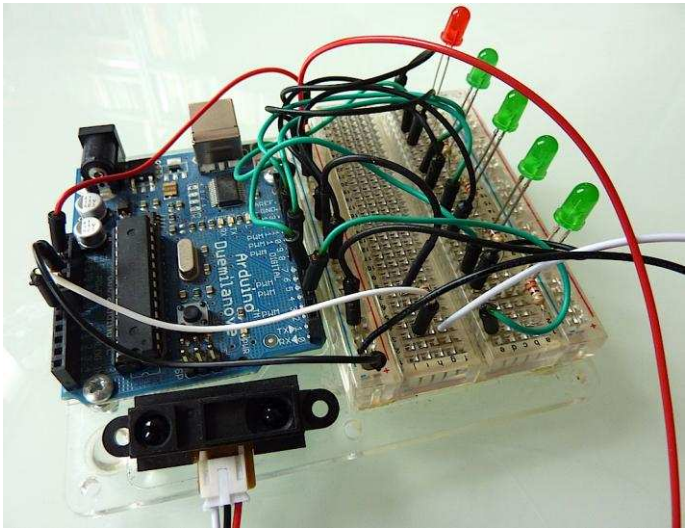

Fig. 1. Example of a prototype using an Arduino microcontroller board

2 Related Work

A visual programming language (VPL) uses a visual representation instead of, or in addition to, more traditional textual representations of program source code. Examples of VPLs are Forms/3 [4] in the spreadsheet paradigm, LabVIEW [31] for instrument control and industrial automation, as well as Scratch [18] for teaching children how to program using animations.

There have been numerous investigations to substantiate the benefits of VPLs, both generally and in certain application areas. Whitley [20] found that visual notations have potential for making information explicit and providing better organisation,

which may help with program design, problem-solving and performance, even more so as the size of problems increases. However, there is still a lack of evidence that they are generally easier to learn, understand, use and share [2] and in addition it appears that the benefits of a notation are relative to a particular application area (task [8]).

One area where visual notations have been used with great success is in teaching children and young people to program. Scratch uses a building-block metaphor; programs are created by snapping graphical blocks of different colours and shapes together to form "stacks" (procedures). These blocks represent commands, datatypes, etc. and users can choose them from a palette of available blocks. The shape of the blocks determines what they can connect with. In this way syntactically correct statements and programs are encouraged: the shapes of the blocks hint as to what is expected and thus provide a guide to the user as to what is possible. There is a growing body of research into the efficacy of Scratch as a tool for learning to program. Several studies report that children and young people - both boys and girls - find Scratch engaging and fun to use, while successfully familiarising them with fundamental programming concepts without the distraction of syntax [7, 13, 14, 16].There is also evidence of continued engagement with computer science. A positive first experience with programming in an environment with a low barrier to entry, like Scratch, can sustain enthusiasm and ease transition to more sophisticated languages, such as Java and C [13, 21]. The visual programming environment used in our study, Modkit [27], was heavily influenced by Scratch and uses a similar building-block metaphor. Modkit has been used in workshops teaching children how to program Arduino [17]. Yet while Scratch has been studied to some extent, so far there are no similar studies to investigate the benefits of ModKit or other VPLs for the Arduino.

There are also some concerns about the habits Scratch engenders [15]. Following a constructivist philosophy of learning-by-making, Scratch both supports and encourages a bottom-up approach to program construction – programming by bricolage [19], rather than design. This can lead to a trial and error approach of programming with extreme decomposition and, in turn, this can make the code highly concurrent and difficult to debug. However, this type of opportunistic construction – tinkering and experimenting – is often characteristic of prototyping that involves electronic components [3, 10], so an environment explicitly designed to support this style of program creation may provide a good route into programming Arduino. Our work is the first step to investigate barriers that end users encounter in the area of programming of physical prototypes.

3 Study Design

We recruited participants for our empirical study through the MzTEK [30] and London Hackspace [25] mailing lists. We used a within-subject, think-aloud design, comprising eleven participants (8 female, 3 male, mean age 36.18), exposing each participant to both a visual and textual environment. None of the participants had previous experience of programming an Arduino or were professional programmers

but they were required to have some previous programming experience (declaring and using variables and 'if statements' in at least one procedural programming language).

Participants used both a textual and visual environment for Arduino programming in our study. The textual environment was the default Arduino programming environment v1.0.1 [24] (Figure 2), which contains a text editor, a message area (for compiler messages), a text console (for text input and output during runtime), a toolbar containing frequently used commands and various menus for operation.

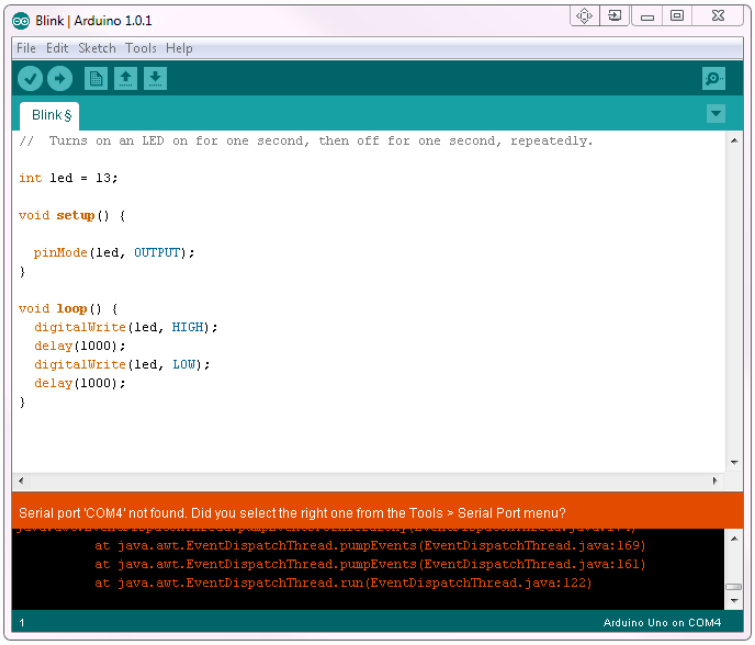

Fig. 2. Arduino IDE (textual programming environment)

The visual environment used was Modkit Alpha Editor (preview version) [28]. It supports multiple code representations, with both a Blocks view (Figure 3), where users manipulate coloured, graphical blocks representing code, and a Source view, which provides a textual representation of the current program(s), also editable. The Hardware view allows users to configure hardware setup visually. While there are a number of other visual programming environments for Arduino, we chose to use Modkit for this study because in terms of terminology, commands available, feature set and code constructs it matches the Arduino language and IDE most closely, thus facilitating isolation of the visual component to the programming experience.

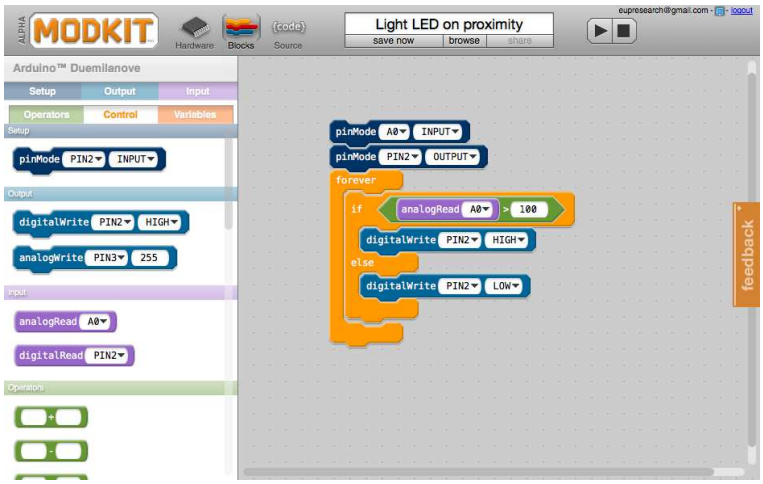

Fig. 3. Modkit Alpha Editor IDE - Blocks view (visual programming environment)

Study sessions took place in the Interaction Lab at City University London, lasting approximately 2 hours per participant. Each session was video-recorded using Morae [29], capturing all on-screen activity, verbal comments and non-verbal behaviour. Mouse-clicks, keystrokes and mouse movements (in pixels) were also logged automatically. Participants were asked to 'think aloud', to provide us with insight into their thought processes.

At the start of the session participants completed a background questionnaire and semi-structured interview regarding their previous programming experience. We then familiarized the participants with the Arduino platform and key Arduino programming concepts and constructs. The tutorial also included a printed hardcopy of a list of key commands, which the participant was able to refer to during the whole session.

For the main part of the study, participants completed two tasks (counterbalancing was used to counter any order effects of environment). In task 1 they were asked to modify an existing program, extending it to meet new requirements. The original prototype contained a single LED and a proximity sensor. The goal of the original program was to light the single LED when an object reached a specific 'closeness' to the proximity sensor. The second prototype extended upon this with the addition of four more LEDs. Each LED was associated with a specific proximity threshold and only when that threshold was reached must the LED be lit. When an object reached the 'closest' threshold, all of the LEDs must be lit. In task 2 they were asked to create a new program from the ground up. The participant was shown a prototype that contained a single LED and a tactile switch. The program must light the LED when the switch is pushed down. Hardware prototypes of the circuits required for each task were prepared in advance; participants therefore did not have to understand and construct electronic circuits. The facilitator demonstrated the desired results using the hardware prototype so that the participants were able to view the working prototype and desired result. Participants were given twenty minutes to complete each task.

We captured two sets of data to investigate efficacy and user experience. We asked participants to think-aloud while they were completing the tasks and we conducted a qualitative analysis of transcribed video recordings for task 2, based on the set of Learning Barriers [11]. The coding scheme used in the study is given in Table 1. Following each task, each participant was presented with a set of 92 word reaction cards, based on the Microsoft Desirability Toolkit [1], capturing their user experience in a qualitative way.

Table 1. Coding scheme used to identify learning barriers in programming the Arduino.

Code	Applied when…	Example application in this study
Design barrier	The user does not know exactly what they want the computer to do.	A user not knowing whether they need to connect the tactile switch pin as input or output.
Selection barrier	The user knows what they want to do, but does not know which tool to use.	A user not knowing which block to use to achieve a particular goal, which operator to use in a conditional statement, or which command to use to read the value of a digital pin.
Coordination barrier	The user knows what tools to use, but not how to make them work together.	A user not knowing how to use blocks or functions in conjunction with one another. The user may know that they need to declare variables and read pins, but not how to get them to work together.
Use barrier	The user knows what tools to use, but does not know how to use them properly.	A user not knowing how to use `pinMode` to set input or output for a digital pin, or how to use the `digitalWrite` command to write a value to one; also not knowing how to declare variables or use an if-else statement.
Understanding barrier	The user thought they know how to use something but it did not do what they expected.	A user unable to correctly interpret a compiler error or understand why something did not happen when it was supposed it, e.g. an LED does not light up when a switch is pushed.
Information barrier	The user has an idea or hypothesis about why their program did not do what they expected, but they do not know how to check.	A user not knowing how to use the serial monitor for debugging, or whether there is a code verification tool they can use.

We used two quantitative measures to evaluate potential benefits and costs. First, we used a self-efficacy questionnaire [6] which gathered participants' levels of confidence with regards to completing programming tasks in programming environments. Participants filled in this questionnaire at the start of the session prior to the participant undertaking any task and then following each of the two tasks. The initial self-efficacy questionnaire provided a baseline score against which post-task self-efficacy could be compared. Further, the NASA-TLX questionnaire [9] was completed by each participant following each task, which measured the perceived load of a task in terms of mental, physical and temporal demand, performance, effort and frustration.

4 Results

4.1 Effects on End-User Programming Efficacy

We wondered if using the Arduino VPL would have any benefits in helping end-users to program compared to the textual programming environment. We found that task completion rates were low in both environments. Only two participants out of eleven completed both tasks that were set in the study, two completed one task and seven did not complete any task. However, the four participants who completed tasks were more successful using the visual programming environment: only two tasks were completed using the textual environment whereas four tasks were completed using the visual environment (Figure 4, left).

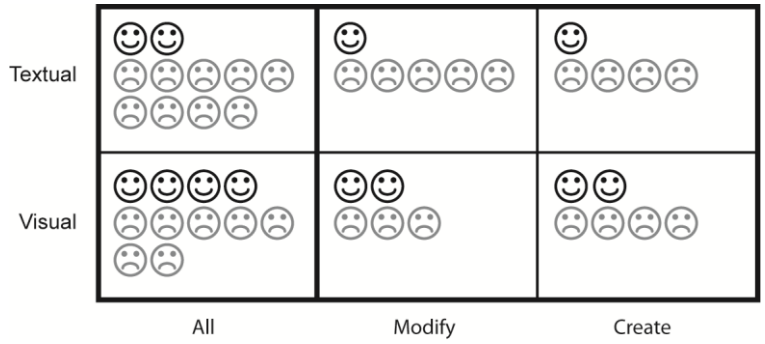

Fig. 4. Participant completion using programming environments for all tasks (left) and for create/modify task types (right)

Most of the problems, which we explore more fully in section 4.3 Learning Barriers, that seemed to prevent completion, were due to difficulties with syntax in the textual environment:

"I just can't remember what all the little coding mean [sic], you know, whether it's an exclamation mark or an equals and then the curly brackets".

One explanation for the low task completion scores in the textual environment may therefore be that participants were focusing more on syntax rather than on program

design. As participant 1 reflected: *"I was too busy worrying about where it should go* and not what was actu*ally in it".*

We also noticed that the kinds of tasks mattered: visual programming helped slightly more in the modification task than the creation task (Figure 4, right) Again, this difficulty may have been compounded by the visual programming environment, which may encourage less focus on overall design and only encourage a trial and error approach [19].

This suggests visual programming environments may provide a slight edge for successful programming over textual programming environments, if users have relatively little programming experience. However, visual programming may be most helpful if users do not have to write the program from scratch.

4.2 Effects on User Experience

In addition to contributing to actual programming efficacy, the type of programming environment may lead to perceived benefits which in turn can determine preference and encourage continued use. We measured the effects on participants' experience through three aspects: their perceived workload and success, their self-efficacy ratings and their reactions to the two programming environments.

Participants in our study generally rated their perceived workload as higher in the textual environment, although they rated the visual environment as more physically demanding (Figure 5; a higher score indicates either higher workload or decreased performance). Although this may seem initially perplexing we found that participants carried out vastly more mouse clicks and mouse movements in the visual environment, which may explain this perceived cost.

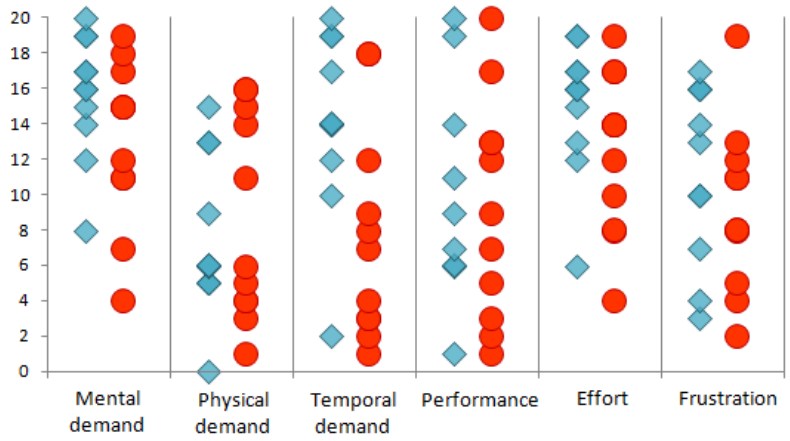

Fig. 5. Participants' TLX scores for textual (diamonds) and visual (circles) environments. The mean score was always higher for the textual environment, except relating to Physical demand.

Furthermore, as the performance scores show, in addition to being slightly more successful in actual programming efficacy, participants also perceived themselves to

perform better using the visual programming environment. This effect on perceived efficacy was also reflected to some extent by participants' self-efficacy scores, which suggest that these benefits may carry over to future tasks. We observed that while self-efficacy scores improved from their initial self-assessments (mean 7.26) in both environments (mean textual 7.38, mean visual 7.90), participants rated themselves as slightly more capable of completing similar tasks after encountering the visual environment. This is summarized well by the following comment from participant 1:

"I think I would quite happily play with that one [visual] and see if I could get something going and i*f I got stuck…but I'm really confident I could make som*ething work without getting stuck*, and, you know, be able to build from there."*

Task type mattered in participants' self-efficacy rating (Figure 6) and echoes our findings for observed efficacy. Participants felt more confident modifying a program using the visual environment than creating one. In contrast, when they employed the textual environment the reverse tended to be true; creating programs made them feel more confident than modifying existing ones. As program reuse is a common strategy for novice programmers, visual programming environments may therefore provide an initial boost to learning by increasing an individual's confidence in their ability to successfully modify existing programs.

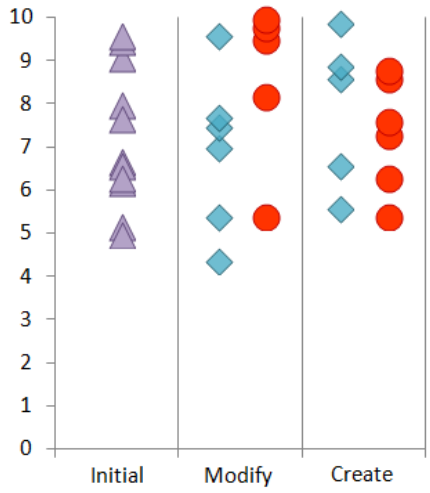

Fig. 6. Participants' self-efficacy scores initially (triangles) and after completing a task using the textual environment (diamonds) and visual environment (circles). Mean score for visual environment is higher than textual environment in the modify task but lower in the create task.

We also noted differences in the type of reactions to the environments. From the set of word reaction cards, to describe the textual environment, participants chose 62 positive words and 69 negative words. This means that they had a fairly balanced experience. Contrast this to the visual environment which received 101 positive word descriptions and only 37 negative words – participants viewed the visual programming environment much more favourably.

Looking at the meaning of the words that participants chose provides further insight about qualitative differences in user experience. Figures 7 and 8 show a visual representation of the words participants selected, their frequency (size) and type (positive vs negative). On the positive side, participants found the visual environment 'fun', 'attractive' and 'easy to learn', whereas they found the textual environment 'clean', 'familiar' and 'understandable'. This suggests that the visual environment is much more emotive, particularly for someone beginning to program, whereas the textual environment is seen as more practical, as this comment from participant 1 suggests:

"It looks the same as every other blank bit of paper programming environment. But I know t*hat that's efficient."*

On the negative side, the main complaint in the textual environment was that it was 'unsupportive'; the visual programming environment was seen as 'confusing'. Part of the challenge in the visual environment may have been due to the "puzzle-like" nature of the screen elements which lock together, mentioned by both participant 9 and 4:

"At the same time, for me, it's like a puzzle on top of a puzzle."

*"You feel like you're solving a puzzle here, but yeah, I'm not sure what the sol*u-ti*on to the puzzle is."*

Fig. 7. Textual environment reactions, positive (in grey) and negative (in black)

Fig. 8. Visual environment reactions, positive (in grey) and negative (in black)

When asked in the post-session debriefing interview which of the environments they would choose to use for learning to program Arduino, the participants were overwhelmingly in favour of the visual environment, with seven of the eleven choosing it. However several participants commented that graphical and textual representations of the same program – like the Blocks view and the Source view in Modkit - might prove useful for different activities or stages of a programming project.

4.3 Learning barriers

We wanted to explore the reasons for some of the negative user experiences reported and also how to facilitate improvements to these programming environments. We therefore looked at what learning barriers our participants faced in the creation task (recall that this was the more "difficult" task to complete). Recall that we randomised the order of programming environments over tasks; this means that six participants encountered the visual environment whereas only five used the textual environment in the creation task. On average, participants encountered 16.82 learning barriers (15.4 textual, 18 visual). Figure 9 shows the percentage of occurrences of each learning barrier in the respective environments; in the remainder of the paper we show raw counts of barriers encountered. In our analysis we will focus on four learning barriers – Use, Understanding, Coordination, and Selection – because this is where the main differences between the environments occurred.

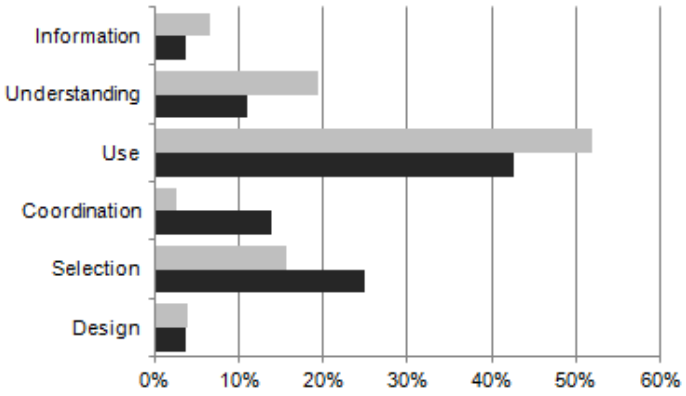

Fig. 9. Percentage of learning barriers observed during the create task for texual (grey) and visual (black) environment.

Use barriers proved to be a big challenge for participants but more so in the textual environment (40/77 textual, 46/108 visual). As already mentioned in section 4.1, a lot of instances for this type of barriers were caused by a lack of familiarity with the syntax of the textual notation, for example:

"If, then, else… It's a standard, conditional thing which you use in every single programming language, but it's always different. It always appears to come with *some extra thing which you don't know about."* (Participant 2)

This barrier is somewhat lessened by the visual environment but still, both environments could provide users with better instructions for correct usage, possibly through context-sensitive instructions which some other IDEs already provide. (The most recent version of Modkit - Modkit Micro - now provides tooltips).

Similarly, understanding barriers encountered in the textual environment outnumbered those found in the visual environment (15/77 textual, 12/108 visual), as this quote from participant 3 demonstrates:

"So it's doing output of zero and one. Oh, ok. So now it's obvious that the problem is that [squints at the screen and shakes her head slightly] it needs some sort of delay. *But that still doesn't make sense, because it shouldn't be doing anything; it should just be in an off state. So why is it flashing on and off?"*

This suggests that the textual environment could provide more support to users in understanding information received at runtime or on compile, and making them aware how to check any hypothesis they may have about the reason for a compiler or runtime error, including the facilities available for testing.

Coordination barriers proved to be a more challenging aspect for participants in the visual environment than the textual environment (2/77 textual, 15/108 visual). Several of these related to the unsuccessful docking together of blocks, which some participants struggled with, suggesting that this interaction could perhaps be improved. In some cases participants were not sure whether blocks had connected, for example participant 5 remarked:

"Did it fit? Yes…No… I'm not sure if it's fitting in the loop, in the If case. It doesn't look like it does."

Some participants, such as participant 2, also did not understand why some blocks did not fit together when they thought they should:

*"What I really want to do is put it there, in the forever loop, [tries to dock digita*l-Read again un*successfully…] but that's not going to happen is it?"*

A frequent barrier for participants in the visual environment proved to be a selection barrier (12/77 textual, 27/108 visual), as this participant stated:

"It's a shame you can't find anywhere like a help or thingy what's… what those [the operator blocks] are*."* (participant 4)

This is somewhat surprising, as we had anticipated that making the blocks explicitly available might make it easier for people to decide what to use. However, it may be that in fact people felt confused because the tool provided both a proliferation of choices yet no clue as which one was the right one – again, a "puzzle on top of a puzzle". Future work could concentrate how to guide users in selecting the right blocks to use.

5 Discussion and future work

Our study suggests that task type, such as creation and modification, play a role in the success of using a programming environment. Participants in our study also felt more confident using the visual programming environment in the modification task. This may explain the effectiveness of using VPLs for teaching: often novices start out

by reusing existing solutions and modifying them – a task for which a visual environment may be better suited. Consequently, there may be an additional "boost" to learners to continue using VPLs for this and similar tasks. Arduino draws in adults with little programming experience and how best to teach these individuals to develop physical prototypes, possibly through 'prototype repositories' which they can reuse, would be an interesting area for further study.

We have started to identify the barriers end users may encounter when programming an electronics platform, however, this work could benefit from comparative studies both between application areas (e.g. intelligent agents, spreadsheets, etc.), as well as other visual programming environments for Arduino, such as S4A (Scratch for Arduino) [33], Minibloq [26], ArduBlock [22] and Modkit. The Blocks view in Modkit, while graphical, uses similar terminology to the default Arduino IDE. Other visual environments, such as Minibloq, differ more radically in their presentation of programming constructs and their approaches to program construction. It would be useful to evaluate these empirically for learning barriers, usability and user experience.

Finally, our participants hinted at the effect of visual and textual environments on programming activities, stating that visual may be good to get them started but that they may want to switch. It could be that debugging in particular is hampered in VPLs: our findings show that Coordination barriers were especially more frequent in the visual environment. One possible solution is to provide both representations at the same time, to provide information to the end user in combination in order to overcome this learning barrier. Future work may be able to explore this solution on programming effectiveness, or how certain programming activities could be better supported in visual programming environments.

6 Conclusion

The overall objective of this project was to investigate whether a visual programming language for Arduino offers benefits over a traditional, text-based language for adult end-user programmers. We learned that:

- Visual environments seemed to help but possible more to modify programs than create them. Further studies are necessary to look into this in more detail.
- Visual environments provided a more positive user experience, alongside a reduced perceived workload and higher perceived success, both for the tasks in the study as well as future tasks. This may entice beginners to continue using these environments even in the absence of actual benefits.
- Both environments are not perfect, and may be perceived as unsupportive or confusing. We showed that this may be due to Use, Understanding, Selection and Coordination barriers. Addressing these may help to improve support for end-user programmers.

Taken together, visual programming languages appear to hold promise for adult novice end-user Arduino programmers but further work is needed to fully understand how best to support end users to achieve both actual and perceived benefits. Our study is a first step in this direction.

Acknowledgements. We thank Ed Baafi and the Modkit team, who kindly gave us access to the Modkit Alpha Editor for use in our study. Thank you also to our study participants.

References

1. Benedek, J., Miner, T.: Measuring Desirability: New methods for evaluating desirability in a usability lab setting. Presented at the Usability Professionals' Association Conference 2002 , Orlando, Florida, USA July 8 (2002).
2. Blackwell, A.F.: Metacognitive Theories of Visual Programming: What do we think we are doing? , IEEE Symposium on Visual Languages, 1996. Proceedings. pp. 240 –246 (1996).
3. Brandt, J. et al.: Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. Proceedings of the 4th international workshop on End-user software engineering. pp. 1–5 ACM, New York, NY, USA (2008).
4. Burnett, M. et al.: Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. Journal of Functional Programming. 11, 02, 155–206 (2001).
5. Cao, J. et al.: End-User Mashup Programming: Through the Design Lens. Proceedings of the 28th international conference on Human factors in computing systems. pp. 1009–1018 ACM, New York, NY, USA (2010).
6. Compeau, D.R., Higgins, C.A.: Computer Self-Efficacy: Development of a Measure and Initial Test. MIS Quarterly. 19, 2, 189–211 (1995).
7. Franklin, D. et al.: Assessment of Computer Science Learning in a Scratch-Based Outreach Program. Proceeding of the 44th ACM technical symposium on Computer science education. pp. 371–376 ACM, New York, NY, USA (2013).
8. Gilmore, D.J., Green, T.R.G.: Comprehension and Recall of Miniature Programs. International Journal of Man-Machine Studies. 21, 1, 31–48 (1984).
9. Hart, S.G., Staveland, L.E.: Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In: Hancock, P.A. and Meshkati, N. (eds.) Human Mental Workload. pp. 239–250 North Holland, Amsterdam (1988).
10. Hartmann, B. et al.: Hacking, Mashing, Gluing: Understanding Opportunistic Design. IEEE Pervasive Computing. 7, 3, 46–54 (2008).
11. Ko, A.J. et al.: Six Learning Barriers in End-User Programming Systems. Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing. pp. 199–206 IEEE Computer Society, Washington, DC, USA (2004).
12. Kulesza, T. et al.: Fixing the Program My Computer Learned: Barriers for End Users, Challenges for the Machine. Proceedings of the 14th international conference on Intelligent user interfaces. pp. 187–196 ACM, New York, NY, USA (2009).
13. Malan, D.J., Leitner, H.H.: Scratch for Budding Computer Scientists. Proceedings of the 38th SIGCSE technical symposium on Computer science education. pp. 223–227 ACM, New York, NY, USA (2007).

14. Maloney, J.H. et al.: Programming by Choice: Urban Youth Learning Programming with Scratch. Proceedings of the 39th SIGCSE technical symposium on Computer science education. pp. 367–371 ACM, New York, NY, USA (2008).
15. Meerbaum-Salant, O. et al.: Habits of Programming in Scratch. Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. pp. 168–172 ACM, New York, NY, USA (2011).
16. Meerbaum-Salant, O. et al.: Learning Computer Science Concepts with Scratch. Proceedings of the Sixth international workshop on Computing education research. pp. 69–76 ACM, New York, NY, USA (2010).
17. Millner, A., Baafi, E.: Modkit: Blending and Extending Approachable Platforms for Creating Computer Programs and Interactive Objects. Proceedings of the 10th International Conference on Interaction Design and Children. pp. 250–253 ACM, New York, NY, USA (2011).
18. Resnick, M. et al.: Scratch: Programming for All. Commun. ACM. 52, 11, 60–67 (2009).
19. Turkle, S., Papert, S.: Epistemological Pluralism and the Revaluation of the Concrete. Journal of Mathematical Behavior. 11, 1, 3–33 (1992).
20. Whitley, K.N.: Visual Programming Languages and the Empirical Evidence For and Against. Journal of Visual Languages & Computing. 8, 1, 109–142 (1997).
21. Wolz, U. et al.: Starting with Scratch in CS 1. Proceedings of the 40th ACM technical symposium on Computer science education. pp. 2–3 ACM, New York, NY, USA (2009).
22. ArduBlock, http://blog.ardublock.com/.
23. Arduino, http://www.arduino.cc/.
24. Download the Arduino Software, http://arduino.cc/en/Main/Software.
25. London Hackspace, https://london.hackspace.org.uk/.
26. Minibloq, http://blog.minibloq.org/.
27. Modkit, http://www.modk.it/.
28. Modkit Alpha Club, http://www.modk.it/alpha.
29. Morae usability testing software, http://www.techsmith.com/morae.html.
30. MzTEK: A learning community in technology and arts for women, http://www.mztek.org/.
31. National Instruments LabVIEW, http://www.ni.com/labview/.
32. Raspberry Pi, http://www.raspberrypi.org/.
33. S4A: Scratch for Arduino, http://seaside.citilab.eu/scratch/arduino.