Combining Tiled and Textual Views of Code

Michael Homer and James Noble
School of Engineering and Computer Science
Victoria University of Wellington
New Zealand
Email: {mwh,kjx}@ecs.vuw.ac.nz

***Abstract*—“Jigsaw puzzle” programming environments manipulate programs primarily by drag-and-drop. Generally these environments are based on their own special-purpose languages, meaning students must move on to another language as their programs grow. Tiled Grace is a tile-based editor for Grace, an educational programming language with a conventional textual syntax. Using Tiled Grace, programmers can move seamlessly between visualising their programs as tiles or source code, editing their programs via tiles or text, and continue on to traditional textual environments, all within the same programming language. We conducted a user experiment with Tiled Grace, and present the results of that experiment showing that users find dual views helpful.**

I. Introduction

Visual programming environments like Scratch [1] present a program as a combination of nested “jigsaw piece” tiles manipulated by drag-and-drop, and have been used successfully with new programmers [2], [3], [4], [5]. These environments present a limited language with a restricted expressive domain, meaning that eventually programmers must move on to a “real” textual programming language and, in many cases, learn to program over again [6], [7]. Tiled Grace is a programming environment for Grace bridging these two worlds: programs may be edited using a drag-and-drop tile interface, but with tiles showing the concrete text syntax. In Tiled Grace, users can switch to a conventional textual view at any time, and can edit that text before switching back to the tile view, making the correspondence between tiles and source code clear.

This paper is structured as follows. In the next section we briefly introduce Grace, and then in Section III describe Tiled Grace and explain the design choices we made in it. Section V describes the additional functionalities we implemented on top of the base system. Section VI describes the user experiment we ran using Tiled Grace with novice programmers, and Section VII the results we obtained. Section VIII positions Tiled Grace among related work, and Section X concludes.

This paper expands upon an earlier short paper from VISSOFT 2013 [8] by incorporating additional functionality and performing a user experiment to validate our design.

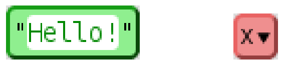
II. Grace

Grace [9] is a new object-oriented language that supports a variety of approaches to teaching programming. Grace integrates accepted new ideas in programming languages into a simple language that allows students and teachers to focus on the essential complexities of programming rather than the accidental complexities of the language.

Grace follows a conventional curly bracket textual syntax and a semantic model that should map cleanly onto almost all other object-oriented languages. To permit different teaching styles a system of dialects [10] allows the definition of sub-languages including new definitions, control structures, and restrictions.

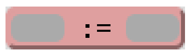
III. Tiled Grace

Tiled Grace presents an editing environment for Grace programs based on drag-and-drop *tiles*. A tile represents a single syntactic unit in the program, such as a string literal, variable assignment, or method request. For example, tiles for a string “Hello!” and variable “x” look like this:


```
"Hello!"        x▾
```

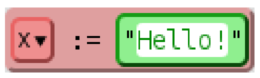
Some tiles, like the string tile above, have text input fields for the user to type a value.

Some tiles have *hole*s in them, where another tile may be placed. For example, a variable assignment tile has two holes: one for the variable to be assigned to, and one for the value:


```
:=
```

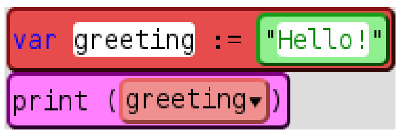
The holes are the empty grey rounded-rectangular areas. Other tiles with holes include operators like + and *, method requests, and print statements. The user can place another tile inside a hole to build up their program.

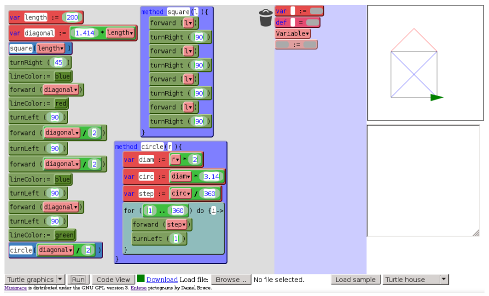
To assign the string “Hello!” to the variable “x”, the user puts these three tiles together in one:


```
x▾ := "Hello!"
```

To put a tile into a hole, the user drags the tile they want to use over the hole, which will be highlighted when they are over it, and drops it there. The hole will expand to fit its new contents.

Tiles can be connected together in sequence as well. To create a variable and print its value, a **var** tile and a print tile can be joined together.


```
var greeting := "Hello!"
print (greeting▾)
```



```
var length := 200
var diagonal := 1.414 * length▾
square(length▾)
turnRight ( 45 )
lineColor:= blue
forward (diagonal▾)
lineColor:= red
turnLeft ( 90 )
forward (diagonal▾ / 2 )
turnLeft ( 90 )
forward (diagonal▾ / 2 )
lineColor:= blue
turnLeft ( 90 )
forward (diagonal▾)
turnLeft ( 90 )
lineColor:= green
circle(diagonal▾ / 2 )
```

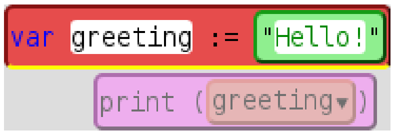
```
method square(l ){
    forward (l▾)
    turnRight ( 90 )
    forward (l▾)
    turnRight ( 90 )
    forward (l▾)
    turnRight ( 90 )
    forward (l▾)
    turnRight ( 90 )
}
```

```
method circle(r ){
    var diam := r▾ * 2
    var circ := diam▾ * 3.14
    var step := circ▾ / 360
    for ( 1 .. 360 ) do {i->
        forward (step▾)
        turnLeft ( 1 )
    }
}
```

Minigrace is distributed under the GNU GPL version 3. Entypo pictograms by Daniel Bruce.

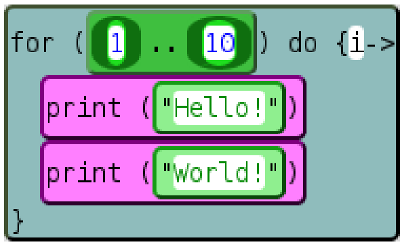
Fig. 1: Tiled Grace editing a small program in the "turtle graphics" dialect.

The user can join tiles together in this way by dragging so that the top of the tile they want to join on is near to the bottom of the tile they want to join onto, which will then be highlighted, and dropping the tile there:



```
var greeting := "Hello!"
    print (greeting▾)
```

Some holes can hold more than one tile, such as the hole in the body of a loop. The first tile can simply be dropped in as for any other hole, and then other tiles can be joined onto the bottom of it. The following code prints "Hello!" and "World!" ten times each in alternation:



```
for ( 1 .. 10 ) do {i->
    print ("Hello!")
    print ("World!")
}
```

A complete program and its output is shown in the Tiled Grace interface in Figure 1. The interface is divided into three main areas: a large workspace area on the left, a toolbox of available tiles, and text and graphical output areas on the right.

Tiles may be dropped anywhere in the workspace pane, and the user can construct different sub-programs in different parts of the area. Different categories of tile can be accessed from a menu in the toolbox. At the bottom of Figure 1 is displayed the dialect selector, run button, and other interface controls.

Different kinds of tile are shown in different colours, with closely related concepts, such as variable declaration, reference, and assignment, having similar colouring.

The feel of Tiled Grace is similar to Scratch [1]. Tiled Grace differs in that it is backed by a genuine textual language: the tiles correspond to the syntax of the Grace language, in order to support students when they eventually move out of Tiled Grace and begin writing textual programs. Tiled Grace goes a step further still: because the tiled representation maps exactly onto text the user can switch between tiles and a standard syntax-highlighted textual view at any time.

The transition from tiled to textual view is shown through a smooth animation. Each tile and block of code has a continuous visual identity throughout the transition, which takes just under two seconds. First the tiles fade out to blocks of the corresponding textual code, then the blocks glide into place in a linear textual program, and finally the display switches to editable text. The entire transition takes just under two seconds. When the user switches back to tiles, the same occurs in reverse. Figure 2 shows this transition in progress.

Each group of connected tiles is regarded as an independent part of the program, and the ordering between them in the textual display is arbitrary, but consistent within the session. This text is editable if the user wishes: they may change the source code, including adding and removing whole lines or blocks, and then transition back to the tiled view.

A. *Implementation*

Tiled Grace is built on top of Minigrace, our prototype Grace compiler, using its JavaScript backend with a new front-end interface. Tiled Grace runs in a web browser without installation, and can be accessed at http://ecs.vuw.ac.nz/~mwh/minigrace/tiled/. Tiled Grace runs in recent versions of Firefox and Chrome, but does not work in other browsers at the time of writing. To execute the code, Tiled Grace generates textual Grace code from the program tree and gives that code to Minigrace to compile, then executes the resulting JavaScript.

IV. Motivation

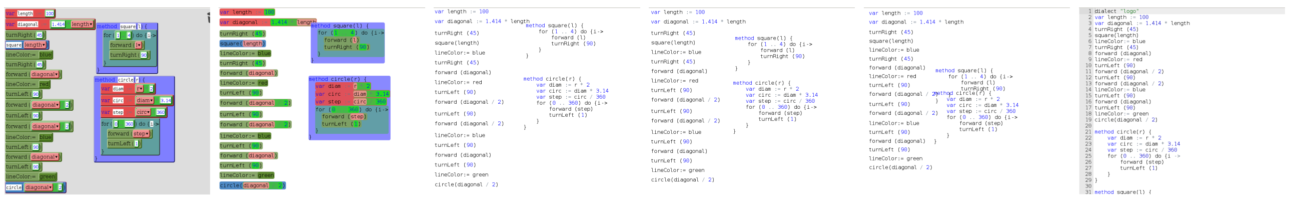
When Scratch, Alice, and similar systems already exist, why build Tiled Grace? Our design goal was to avoid some pitfalls and problems that have been encountered with these existing systems while remaining usable and engaging. In this section we describe the issues with other systems that motivated the different design choices we made in Tiled Grace.

One issue that has been encountered in introductory visual languages is that learners do not see them as "real" programming languages [11], [12], particularly when they move on to textual languages and struggle initially [6]. These students may feel that the visual language "didn't count" and that they are not capable of "real" programming, which view is harmful.

In Tiled Grace we aim to avoid or ameliorate this perception by presenting the textual and visual representation of code equally, and clearly the same language. The textual-tiled combination was our original conception for Tiled Grace.

Another reported problem with moving on from visual to textual languages [6], and moving between languages early in learning in general, is that learners find it difficult to connect analogous concepts in one language to the other. Our animated transition between visual and textual representation aims to demonstrate the exact parallel between the two.

In particular, it is known from both educational psychology generally and computer science education that transitioning between languages early in learning is unhelpful [13]. A course structure predicated on such a transition will likely run into trouble, but introductory tertiary courses in Scratch and Alice move on to "real" languages early, often within the first course, as programs become more complex. Permitting



```
dialect "logo"
var length := 100
var diagonal := 1.414 * length
turnRight (45)
square(length)
lineColor:= blue
turnRight (45)
forward (diagonal)
lineColor:= red
turnLeft (90)
forward (diagonal / 2)
turnLeft (90)
forward (diagonal / 2)
lineColor:= blue
turnLeft (90)
forward (diagonal)
turnLeft (90)
lineColor:= green
circle(diagonal / 2)

method circle(r) {
    var diam := r * 2
    var circ := diam * 3.14
    var step := circ / 360
    for (0 .. 360) do {i ->
        forward (step)
        turnLeft (1)
    }
}

method square(l) {
```

Fig. 2: Frames of the animated transition from tiled to textual view. Transitioning from textual to tiled view shows the same intermediate states in reverse.

both views should avoid this transition, so that learners can begin in (Tiled) Grace, move gradually into (textual) Grace, and continue in that full-strength language as long as required.

One issue with language transitions is that they are essentially "one-way": the learner must apply what they know about the earlier language to the later, but movement in the other direction is restricted. Tiled Grace has a deliberately permeable barrier: a user can use the visual language, the textual language, and the visual language again, even within one program. Allowing movement in both directions necessitated some trade-offs (particularly that the programmer could only switch views when there were no errors in the program), but we considered it appropriate to the goal of the language.

Another key motivation was our dialect system, which has no real parallel in the other visual language systems. Scratch, Greenfoot, and Alice all expose different degrees of complexity appropriate to different levels of development, but only one each. Advanced users of Scratch find the limitations frustrating, but permitting more flexibility can lead to early learners becoming stuck. A key decision in the design of Tiled Grace was that it would support dialects from the ground up, so that learners could move into less restrictive language variants as they went, while staying in the same language and same interface. Again, that integration involves some trade off, but we consider it worthwhile to allow a user to remain within the same fundamental language as long as possible.

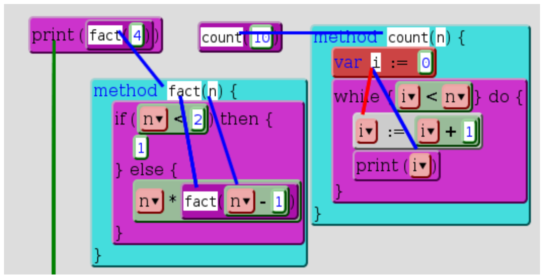
V. Functionality

On top of the basic functioning of Tiled Grace described in the previous section, the tiled view and its duality with the textual representation offer new possibilities for system behaviour. In this section we describe the functionality for handling errors, showing information about definitions, dealing with language variants, and type checking.

A. *Errors, Overlays, and Dialects*

While the tiled view prevents most syntax errors, the user may still write incomplete or incorrect code and these must be reported to the user [8]. A graphical indicator shows whether the program is currently valid; when there is an error the user may hover over the indicator to highlight all existing errors. Error sites are shown by desaturating the code area except the error sites, and overlaying an associated explanation at the site: for example, "Something needs to go in here" at an empty hole.

To prevent errors spreading further than necessary, the user can only switch views when the program is valid. If the user tries to switch while there is an error, the error site will be highlighted and the view unchanged.



```
print(fact(4))
count(10)
method count(n) {
    var i := 0
    while {i < n} do {
        i := i + 1
        print(i)
    }
}
method fact(n) {
    if (n < 2) then {
        1
    } else {
        n * fact(n - 1)
    }
}
```

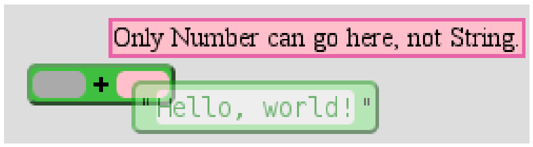
Fig. 3: Composite image of multiple overlays at once.

In the text view, the user is unrestricted in the kinds of error they can produce, as in any textual editor, and errors are reported and marked in the usual way. If the user tries to switch to the tiled view while the program does not compile, they will be presented with the error and asked whether they want to revert to the last-known-good version.

As well as visualising the code itself as tiles, Tiled Grace can visualise relationships between parts of the code [8] (see Figure 3). When a user hovers their mouse pointer over a variable reference, the code view will be overlaid with a line from that reference to the variable's definition site, as well as to any assignments to the variable in scope. Hovering over a variable declaration produces an overlay that indicates all the uses of that variable in scope. Similarly, hovering over a method definition identifies any requests of that method in the program, while hovering over a request (including of a method that came from the dialect) highlights the definition of the method. In this way the programmer can easily read the program in execution order, rather than top-to-bottom, which has been found to be helpful for novices [14]. If applicable, multiple overlays may appear at once. These overlays are similar to those found in spreadsheets [15] to illustrate the dependencies of a formula.

Grace dialects can extend the methods available to the programmer, or provide additional definitions. When the user selects a dialect to use, Tiled Grace creates tiles for all of the provided methods, based on a description of the dialect [8]. This description can be automatically generated from the dialect itself, or manually with additional annotations.

Dialects are an important generalisation of Blockly's ability to choose an extended sub-language to use. Because our



Only Number can go here, not String.

```
+ "Hello, world!"
```

Fig. 4: The display of a simple type error the user is attempting to make, where they try to place a string tile somewhere that only numbers are permitted.

dialects persist and originate textually, the user retains the ability to use and understand them even outside Tiled Grace.

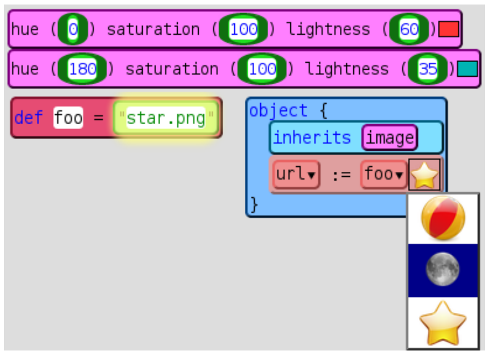
B. *Type checking*

Type checking in a drag-and-drop interface raises additional obstacles versus conventional static type checking. While we can run a standard algorithm over the code and display the results, given the way the user interacts with the system we would prefer to show errors at the time they are made, or even to prevent their occurrence altogether.

We chose to use a variant on our overlay approach to report errors as the user tried to make them, as well as preventing the user from doing so. Any hole, including both those in built-in tiles and those from dialects, can be annotated with the types it will accept, and any tile can be similarly annotated with the type of the object it represents. As Tiled Grace variable declarations do not include static type annotations, all type annotations are currently built in (either to the tool directly or as part of dialect definitions), but the underlying system would need no change to extend to other types were they added.

For example, a string tile is annotated with the type "String", and both holes in a + tile are annotated as accepting only "Number". When the programmer tries to place one into the other, as in Figure 4, the hole is marked in pink and an error message displayed nearby, while the user will not be able to drop the tile into the hole. In this way, the type error is prevented from being introduced into the program in the first place, removing the need for a typechecking pass; nonetheless, some classes of type error could be introduced within textual code and not be caught there, and then make it through the transition to tiled. As a result, the error-handling step described in Section V-A also checks that all holes and their contents are well-typed, and any errors are reported in the same way.

Scratch achieves a sort of type indication through its "jigsaw puzzle" tiles: holes and tiles of different types have different physical shapes, so a Boolean constant or expression will not fit into a numeric expression. We initially wished to use a similar approach, but ran into two problems: a limited number of sensible shapes and difficulty with "multi-type" holes. While Scratch is designed around these shapes and has few types, in Grace we would exhaust the variety of readily distinguishable shapes. At the same time, sometimes we have holes (like equality tests) that can hold multiple types, and shapes alone did not suffice for this situation. Our system provides for an arbitrary number of types (including new types unenvisaged by us), and gives the user explicit feedback and



```
hue (0) saturation (100) lightness (60)
hue (180) saturation (100) lightness (35)
def foo = "star.png"
object {
  inherits image
  url := foo
}
```

Fig. 5: Two hints showing a colour selector (top) and an image preview (bottom). The menu allows changing between known images, and will here update the remote **def** foo.

vocabulary for the error they are having.

C. *Hints*

One advantage of a non-textual display of code, such as our tiled view, is the flexibility to render additional "out-of-band" information within the program display for the benefit of the programmer. In Tiled Grace we call these "hints" and a dialect may define them for its tiles. The dialect we built for graphical programs includes two hints, both showing to the programmer a graphical representation of some text they wrote.

The first hint is on a tile for defining colours using the hue-saturation-lightness scale. A small block of colour appears on the tile, updated live as the programmer edits the values or definitions leading to them. The second involves images: the dialect provides the ability to construct "image" objects, which render an image at run time. The image used is determined by the name assigned to the url field of the object, and the hint catches these assignments, shows a preview of the image referred to, and offers a drop-down menu for the user to select from known images. If the user chooses a new image, the code is updated, including if the original definition site was remote from the code at hand. Both of these are depicted in Figure 5.

These hints are implemented by augmenting a dialect definition with JavaScript functions, which access Tiled Grace's internal representation and API. While the dialect implementor must know the structure of Tiled Grace to build a hint, the end user receives additional help with no effort on their part.

VI. Experiment

Our experiment trialled Tiled Grace with 33 participants, primarily students enrolled in undergraduate courses in the School of Engineering and Computer Science at Victoria University of Wellington. This experiment was approved by the university's Human Ethics Committee. Participants were asked to use Tiled Grace to write, modify, and describe programs, while we recorded their actions. Participants also completed questionnaires about themselves and the experiment.

Our experimental design was guided by some key questions we wished to answer (as well as by practical considerations, particularly timing). We wished to find out whether users

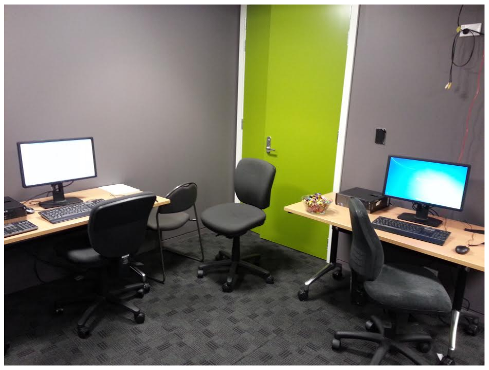


Fig. 6: A photograph of the room used for experimental trials. The two experimental PCs are on the far edges of the picture, with up to one participant on each machine. The experimentor was positioned approximately at the camera during trials.

found the ability to switch views useful, and also whether they appreciated the explicit animation connecting the two, a particular novelty of our approach. We wanted to see whether the error reporting and type checking we had built was useful to users. As a tool that users do not enjoy will not be used, we wanted to measure engagement; finally, we wanted users to explore different parts of the system so we could discover any unanticipated problems or successes.

A. *Experimental protocol*

Participants were recruited by announcements in lectures, forum posts, word of mouth, and direct recruitement, and invited to make an appointment to perform the experiment. These are the standard techniques used for experiments in the department. Participants were able to attend in pairs, with each person performing the experiment simultaneously but independently. Two enticements to participate were provided: a random draw for one of three $50 gift vouchers, and a bowl of assorted confectionery that was available during the experiment and some in-person recruiting sessions.

The experiment was conducted in a room provided by the School of Engineering and Computer Science of Victoria University of Wellington set up for this purpose. The experimental room had two ordinary workstation computers set up, as shown in Figure 6. Each machine had an ordinary keyboard, mouse, and screen, and was running Windows 7. All recorded information, including questionnaires, occurred within a web browser. Google Chrome 33 was used on each machine. The experimentor sat at a distance positioned to see both screens and observed participants during the experiment.

Our experiment involved a tutorial guided by the experimentor and five tasks, all of which involved being presented with a program and some instructions on what to do with it. We selected the tasks with the goal of having users interact with all different parts of the system in mind, while also wishing to have the entire experiment complete within 40 minutes.

While participants used the experimental system their on-screen interaction was recorded by the tool. Every drag, variable selection, text modification, switch of views, or attempt to run the program was noted, and a snapshot taken after every change. These logs were automatically saved to the server while the participant used the system. No audio or video recording was used in the experiment. Participants were automatically prompted to move on after five minutes.

On arrival each participant was led to a workstation with an initial questionnaire open and invited to fill it in. The survey responses were recorded electronically. Following the completion of the initial questionnaire participants were given a scripted tour of the experimental system: we showed a tutorial program in a graphical dialect and demonstrated ways it could be manipulated. After the tour participants could explore the system with the tutorial program before moving on to the first task when they wished. The tasks were:

1) To modify a procedural program printing Fibonacci numbers to print factorials instead. We chose this task to begin with as it could be represented by a single linear block of tiles and involved variable assignments. This program was most similar to simple textual programs from introductory courses.
2) To correct introduced errors in a modified version of the program in Figure 1. The errors were primarily tiles out of place, and a single misspelled method name.
3) To swap behaviours of two graphical objects.
4) To type a description of the behaviour of a program without running it. This program was first presented in the textual view, but users could switch if they wished.
5) A final "task" where users were told they had finished, and could continue to play with the system and move on to the final questionnaire when ready; this task aimed to measure user engagement implicitly. A sample program implementing a crude orbital simulator was given, but users could replace it entirely if they wished.

The final questionnaire asked participants about their interactions with the system and what they preferred. Free-text entry fields were provided with prompts to say what the participant liked or disliked about the system.

In the questionnaire we sought to measure what participants found difficult or easy in the experiment, how engaged they were, which features they had used (particularly the ability to switch views), and what they liked or disliked. Questions primarily asked participants for such information directly and gave a seven-point Likert scale for answers.

VII. Results

A. *Demographics*

33 participants completed our experiment (one further participant withdrew). Participants were principally drawn from students in the School of Engineering and Computer Science at Victoria University of Wellington and so represent at best the demographics of the source. 23 (70%) of participants were male while 10 (30%) were female. The median age of participants was 20 and the most common age was 18. There are decreasingly many participants in older age bands. A full breakdown of these demographics is shown in Figure 7.

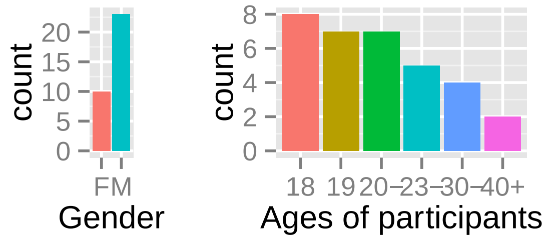


Fig. 7: Basic sample demographics of our experiment.

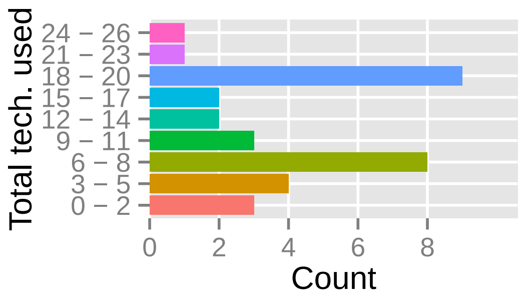


Fig. 8: How many technologies participants had used.

B. Programming experience

We asked questions about past programming experience. The most informative presented 72 technologies (mostly languages, but also IDEs and other tools) and asked participants to indicate any they had used before. The distribution of counts of technologies is shown in Figure 8. The total number of technologies ranged from 1 to 25. The median was 10.

The most popular technologies used were Java and Eclipse (79%), both used in undergraduate courses in the school, while Python (76%) and HTML (73%) were also popular.

Four participants had previously used Scratch, the system most similar to our drag-and-drop interface, while six had used Alice, another introductory programming language with a partial drag-and-drop interface. One had seen Grace.

C. Engagement

A key measure of this system is user engagement. We attempted to measure engagement in multiple ways. In the simplest, we asked participants in the final questionnaire whether the system was fun to use. Responses were on a seven-point Likert scale and shown in Figure 9. Responses 1, 4, and 7 were labelled “Agree”, “Neutral”, and “Disagree”.

The most common response was 2, with nine participants (27%), while 1 (“Agree”) and 3 were chosen eight times (24%) each. 25 participants in total (76%) chose one of the responses on the Agree side. One participant chose 5, a light disagreement, while seven (21%) were neutral. The median response was 2, a medium agreement. We also asked participants for their agreement with “I would use this system again”, and again 76% chose an agreeing response.

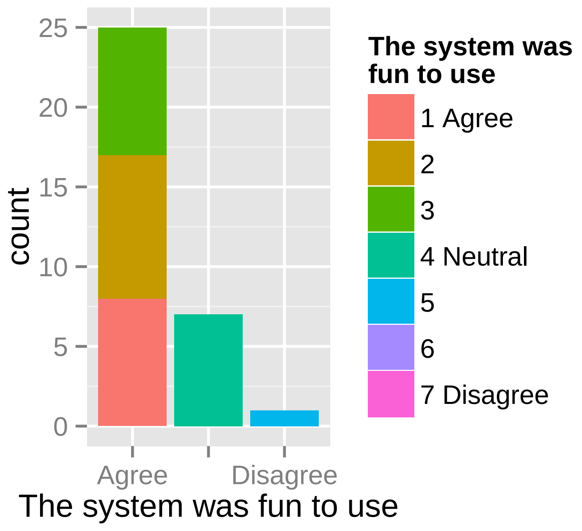


Fig. 9: Participant responses to the statement “The system was fun to use”.

The fifth task of our experiment included a program but no actual task, instead informing participants that they were finished, that they could use the system there if they wished, and to move on to the final questionnaire when they were ready. By this we intended to measure implicit engagement: would participants use the system unprompted? We measured whether participants interacted with the system for 45 seconds or more. We chose this threshold conservatively, allowing 30 seconds for participants to read the task description, look at the program, and potentially run it before moving on to the questionnaire, and adding a 15 second buffer. 23 participants (70%) used the system for at least 45 seconds here, while 10 (30%) moved directly on to the questionnaire. The median time spent here was 1:43 and the mean 3:10.

We take from these results that participants were reasonably engaged with the system. Large majorities in every case indicated some degree of engagement, including both when explicitly asked and through revealed preferences.

Not all participants were as enthusiastic, and we note one trend shown in Figure 10 in particular. If we recall the list of technologies we asked participants about their use of, we can divide participants into two groups: those who have used more than the median number of technologies (16 participants, or 48%), and those who have not (17 participants, or 52%). We can then examine the proportions in each group giving each response to the statement “The system was fun to use”. On doing so we see that participants with less experience are substantially more positive than those with more. Fully 41% of less-experienced participants fully agreed with the statement, while only 6% of more-experienced participants did so. Similarly, 31% of more-experienced participants were neutral, while only 12% of less-experienced participants were. From these responses and regression analysis it appears that

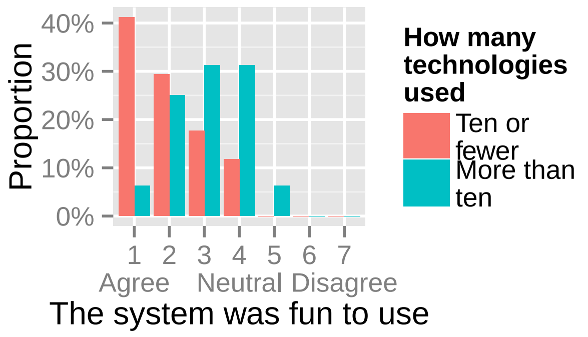


Fig. 10: Participants' agreement with "The system was fun to use" split by how many technologies they had used.

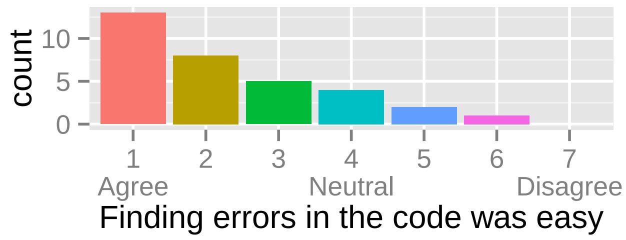


Fig. 11: Participant agreement that finding and fixing errors was easy.

all other things being the same a more experienced user will enjoy the system less. This result is consistent with our and others' experience with Scratch, and not a significant issue for a tool designed for introductory programming.

D. *Error handling*

The tiled interface both prevents some kinds of error from occurring at all and provides the opportunity for entirely new kinds of error. Tiled Grace includes novel error reporting for such code, as described in Section V-A. We asked participants whether finding errors in the code was easy, and also whether fixing them was easy. The results are shown in Figure 11. Responses were on a seven-point Likert scale with responses 1, 4, and 7 labelled "Agree", "Neutral", and "Disagree".

Most participants agreed that finding errors was easy. The modal answer was 1 ("Agree"), with 13 participants (39%), while 26 in total (79%) gave an answer on the Agree side. The median answer was 2, a moderate agreement. Responses were much more varied on the question of fixing errors, with every response from 1 to 5 being chosen by between five and seven participants. Fixing errors in an unfamiliar system, language, and codebase under time pressure would not generally be expected to be easy, so this result is not surprising.

E. *View switching*

Tiled Grace permits switching between tiled and textual views of code at any time. We measured participants' use of this feature and asked them several questions about it.

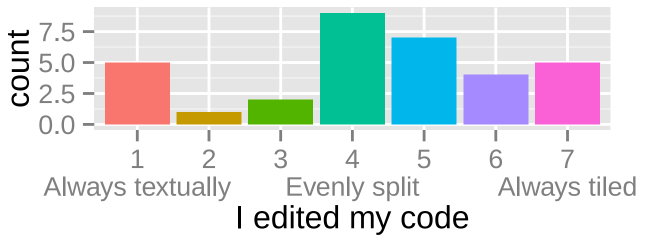


Fig. 12: How participants felt they had edited their code.

	Stat.	T. 1	T. 2	T. 3	T. 4	T. 5	Tot.
Prop. of time in text view	Min.	0%	0%	0%	3%	0%	1%
	1Q	0%	0%	0%	65%	0%	24%
	Med.	17%	53%	8%	100%	0%	33%
	3Q	50%	84%	32%	100%	17%	52%
	Max.	83%	94%	76%	100%	83%	78%
Number of switches of view	Min.	0	0	0	0	0	0
	1Q	0	0	0	0	0	4
	Med.	1	1	1	0	0	6
	3Q	3	3	3	1	2	11
	Max.	8	8	8	5	14	22

TABLE I: Summary and distribution statistics for the usage of different views per Task and overall.

One particular focus of the tiled interface was the elimination of basic syntax errors like mismatched brackets or using the wrong symbol. We asked participants whether they found the syntax easier to deal with in the tiled view. Most participants (18, 55%) chose an answer on the Agree side and answers were steadily less common moving towards Disagree.

We asked participants how they had edited their code. Most participants felt that they had used the tiled view a substantial part of the time, with 16 (48%) indicating they used it most of the time and a nine (27%) saying they had split their time evenly. Five participants (15%) said they used the textual view exclusively. These results are shown in Figure 12. Responses were on a seven-point Likert scale with 1, 4, and 7 labelled "Always textually", "Evenly split", and "Always tiled".

Table I shows the distribution of view switches and time in text mode for each task and overall. The median number of switches is six, the first quartile is four, and the third quartile is eleven. Participants varied significantly in their use of the view-switching feature, using it between zero and 22 times.

Most participants used the tiled view a majority of the time, but most switched views at least once for each of the first three tasks, and used the text view a nontrivial amount of time. These counts and proportions are fairly consistent across tasks until the fourth. This task asked participants to describe a program initially presented as text, and the majority of participants did not switch to the tiled view at all. It may be that participants simply did not think to switch views; an alternative possibility is that they find text more useful for comprehension, but the tiled view helpful for editing code. We will examine these possibilities more closely when analysing the freeform text responses from participants.

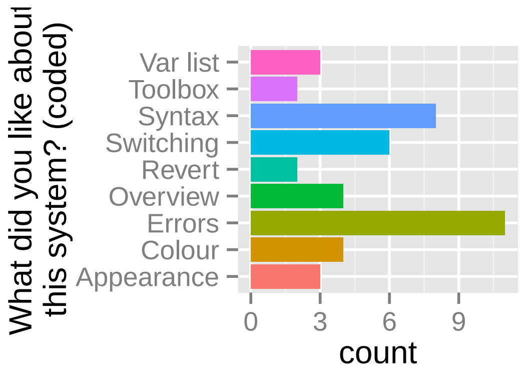


Fig. 13: Coded participant responses to "What did you like about this system?" Any point with more than one mention is included.

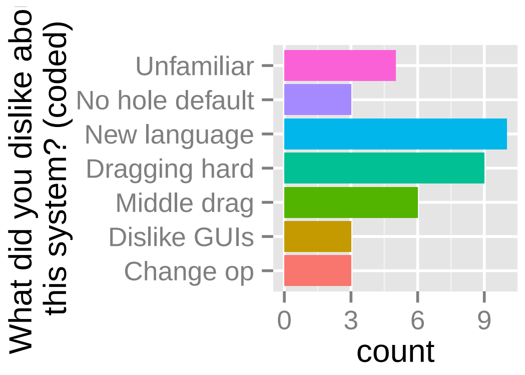


Fig. 14: Coded participant responses to "What did you dislike about this system?" Any point with more than one mention is included.

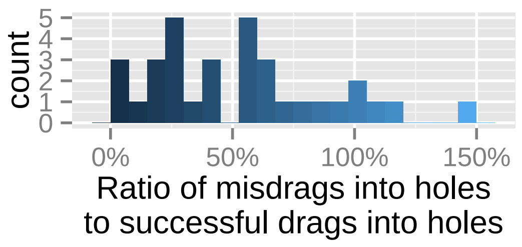


Fig. 15: Ratios of hole misdrags to non-misdrags.

F. *Freeform responses*

We prompted participants for freeform responses on what they liked and disliked about the system. Participants could write arbitrary text in response to these questions. We coded participants' responses to the like and dislike questions and show the distribution in Figures 13 and 14.

Figure 13 shows the distribution of coded responses to "What did you like about this system?". Participants could mention multiple topics and be coded for each. Any mention of the relevant topics was coded into that category. The figure shows all topics that were mentioned more than once.

The most common response was that participants liked the error reporting described in Section V-A and found it helpful. The most interesting response for this experiment was whether participants liked switching views, which six participants identified explicitly, while four said they found the tiled view helpful for an overview of the code and eight found the tiled view helpful for dealing with syntax.

Figure 14 shows the distribution of coded responses to "What did you dislike about this system?". The figure shows all topics that were mentioned more than twice.

The most common dislike was that the drag-and-drop was too sensitive or insensitive or did not do what participants wanted. A common note in the responses that we also observed during the experiment was that some participants found it difficult to drag a tile into a hole. The system required the mouse pointer to be inside the borders of the hole to consider the drag to be over the hole, and not just a portion of the dragged tile; the hole would be highlighted (yellow) when the pointer was over it and a tile was being dragged. We considered this standard behaviour for drag-and-drop and did not give it any significant design thought before the experiment.

We have confirmed subsequently that the default behaviour of the standard interface widgets on Windows, Mac OS X, KDE, and GNOME conforms to this expectation; nevertheless, multiple participants had repeated difficulty here. It may be that this convention is in fact unintuitive and users need to learn it separately for each tool they use. Past human-computer interaction research [16], [17], [18] has found that point-and-click interfaces may involve fewer errors and be faster than drag-and-drop. We examined the interaction data we collected in more detail to try to discover any trends in the data.

We analysed the actions participants took during the experiment to count these "mis-drag" events. We defined a misdrag as a drag and drop onto the background followed immediately by picking up the same tile in a subsequent drag event, without interacting with the system in any other way in between. We defined a "hole misdrag" as a misdrag where the tile was eventually placed into a hole, and an "unrealised misdrag" as one where the participant either tried to run the program or viewed the error overlay immediately after the misdrag, and so could be assumed not to have realised that the tile was not where they wanted. The number of hole misdrags ranged widely from 1 to 36. We can compare these counts to the number of successful drags into holes by the user (Figure 15). Five participants (15%) had more hole misdrags than successful drags into holes, indicating serious difficulty. If we consider unrealised misdrags only we see that again around 15% have difficulty, with all other participants having no unrealised misdrags. One participant had ten unrealised misdrags, while four others had between two and four.

The fact that most participants had at least 10 hole misdrags, and that some participants had debilitating difficulty, suggests that drag-and-drop may be a problematic paradigm for programming. We discuss this issue further in Section IX.

VIII. Related work

A. Scratch

Scratch [1] is a wholly visual drag-and-drop programming environment with jigsaw puzzle–style pieces, aimed at novices and children. Scratch is purely visual: there is no textual representation of Scratch code at all, and some tiles in the system take advantage of layout tricks not possible in text. A Scratch program is able to talk exactly about the graphical microworld the system presents, and no more, so eventually a student must move on and use a "real" language when their programs become more complicated.

Tiled Grace avoids this immediate need by allowing arbitrarily complex programs and always providing an equivalent (and co-equal) textual representation for a program. A student may gradually use the textual editor more and more until they are confident in moving to a more standard environment, or even continue to use Tiled Grace indefinitely without any loss.

In our experiment we found that participants appreciated having a conventional textual view available, even when they preferred to edit graphically. We believe from these results that including a bijective textual representation of code is helpful in visual editors and that Scratch and others should consider incorporating such a representation.

A number of aspects participants in our experiment disliked were common across most tile-based editors including Scratch, notably finding dragging to be a chore, which conformed with our own experience in Scratch. We noted in Section VII-F that some human-computer interaction research has found drag-and-drop to be a suboptimal interface paradigm, and that point-and-click may be superior. We discuss this issue further in relation to both Scratch and Tiled Grace in Section IX.

Scratch includes one notable feature that our system does not: when a Scratch program is running, each tile is highlighted in turn as it is executed. The idea behind this highlighting is to make the flow of control clear, particularly the fact that multiple threads of control flow are executing simultaneously in a Scratch program. Our system does not include such highlighting; primarily, this omission is a technical limitation of the JavaScript environment and the generated JavaScript code from Minigrace. Because Minigrace generated JavaScript code, and browsers execute JavaScript in a single-threaded and blocking fashion, we could not provide any visual update from a program until it completed. Alternative code generation techniques allow solving this problem, but we did not implement these in Minigrace.

B. Blockly

Blockly [19] is very similar in ethos to Scratch. Blockly runs in a web browser and incorporates language variants (what we call *dialects*), but in mimicking Scratch also has no editable textual format. The same limits apply to Blockly and Scratch.

Our experiment found that a bijective textual representation of source code was helpful to users. Blockly supports exporting code to a number of languages, but there is no explicit indication of which parts of the visual representation correspond to which parts of the exported code. Tiled Grace makes this connection clear through animation, and experimental participants indicated that they liked and understood the correspondence. We believe that making the connection between the two formats explicit is important for participants transitioning from visual to textual programming.

C. Alice

Alice [20] is a 3D microworld language manipulated by drag-and-drop. Alice uses drag-and-drop both for putting 3D models into the microworld and for editing logic; there is no interaction with concrete textual syntax. Our system does not include a persistent microworld and does not permit manipulating the worlds it does present (through dialects) other than programmatically. Alice programs can only interact with this microworld and cannot express tasks outside of it.

Event handlers on Alice's in-world objects are put in place through drag-and-drop in a similar way to our tiled view, but there is no editable text. One notable difference in the way the drag-and-drop logic behaves compared to ours is that Alice code does not allow even temporary syntax errors: when placing an "if-then" into the code, the programmer cannot move on to any other task before they fill in the condition. We consider such a prohibition to be a reasonable option, but note that it obstructs other idioms. In particular, one way of programming with both Tiled Grace and Scratch is to drag multiple tiles from the toolbox onto the workspace when knowing that they will be needed and then assembling them once all are available, avoiding back-and-forth trips to the toolbox. We are unsure which approach is best, but a future experiment could use both.

Powers, Ecott, and Hirshfield experimented with transitioning from Alice to Java (with BlueJ) in an introductory programming course [6]. They observed that many students

> were intimidated by the textual language and syntax, and seemed to have a difficult time seeing how the Java code and the Alice code related

even when working with exactly corresponding Alice and Java code. The authors identify this problem as a potential issue for visual programming languages for novices in general. Our system aims to ease this transition to conventional syntax by explicitly showing how tiled and textual code relate. In addition, Tiled Grace was explicitly designed with a permeable barrier in mind: a user is not forced to move entirely into the textual world at once, but can acclimatise gradually.

IX. Future work

While Tiled Grace includes simple type checking to prevent common errors, we would also like, if possible, to signal what is permissible in advance by some feature of the tiles themselves. Scratch and Blockly use a "jigsaw puzzle" approach, where only tiles that "fit" can be placed in any given position, but this is incomplete; some tiles may be the correct shape but still not allowed (in Blockly) or not sensible (in Scratch) in a particular location. We plan to investigate variations of shape, colour, and other attributes to indicate these restrictions in advance of a user trying to perform the task in the program.

The graphical design of the tool would benefit from further consideration. The current colouring of tiles is essentially arbitrary, while the overlays are functional but may obscure areas of the program. The colours used in the interface are not ideal for conveying semantic meaning and should only be used in addition to other indicators. We intend to create a more consistent design and investigate variations to the overlay displays such as transparency and alternative pathfinding.

At present our view-switching system only allows programs with no current errors to be switched to the other view. In part this is for technical and representational reasons: some erroneous code has no clear representation in one view or the other. Some errors, however, could be seen on both sides of the divide, and users may benefit from being able to look at them in two different ways. In future we may allow at least some classes of error to pass through the barrier between the two views, but establishing which errors are suitable, both technically and in terms of not creating additional confusion for the user, is design work remaining to be performed.

In Section VI we outlined experimental results suggesting that some participants had substantial difficulty with drag-and-drop, and many noted some degree of difficulty, suggesting that drag-and-drop may not be the most suitable paradigm for programming. After the experiment we examined the human-computer interaction literature and described in Section VII-F that some HCI research has suggested that drag-and-drop is a problematic interaction mechanism in general, and that a point-and-click arrangement is less error-prone. Further research is required to determine the impact of this issue in relation to visual programming; in particular, given the target markets of Scratch and Grace, and the recent work of Barendregt [21] on children's interaction with various interfaces, more structured classroom-style experimentation may be in order.

X. Conclusion

Tiled Grace is a graphical editing environment for Grace, inspired by visual program editors such as Scratch. Tiled Grace visualises code as nested "tiles" that can be manipulated by drag-and-drop, eliminating many syntax errors. Tiled Grace's tiles always correspond exactly to Grace's textual syntax, so that users become familiar with the textual syntax while dragging and dropping tiles. The user can switch between the tiled and textual view, with the program editable in both forms. Tiled Grace can also visualise relationships between definitions and uses of variables and methods.

We conducted an experiment to measure user engagement with Tiled Grace, and how people would use the tiled view, view-switching and error-reporting provided by the tool. We found that participants generally (76%) enjoyed using our system and that other measures of engagement were high, supporting the use of these features in development tools. We also found that enjoyment was lower for more experienced users, suggesting that Tiled Grace and similar interfaces may be most appropriate within the novice-user market that Grace aims for.

The error reporting (desaturating all non-erroneous tiles and overlaying explanations) was very well received. 79% agreed that finding errors in the code was easy with this reporting style. This approach does not strictly require a tiled view and might have application in more conventional editors as well.

We showed in the experiment that participants found having a more conventional textual view of code available to be helpful, even if they liked to edit the graphical version, and that they liked to have the graphical version available for an "overview" even when they were editing textually. The direct equivalence between the two views was helpful.

Participants also noted features afforded by the tiled view, such as colour coding, a toolbox of available methods, lists of variables in scope, and direct indicators of the definition or usage sites of variables and methods to be helpful. Several of these features could be incorporated into conventional editors.

References

[1] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009.

[2] D. Franklin, P. Conrad, B. Boe, K. Nilsen, C. Hill, M. Len, G. Dreschler, G. Aldana, P. Almeida-Tanaka, B. Kiefer, C. Laird, F. Lopez, C. Pham, J. Suarez, and R. Waite, "Assessment of computer science learning in a Scratch-based outreach program," in *SIGCSE '13*.

[3] Q. Burke and Y. B. Kafai, "The writers' workshop for youth programmers: Digital storytelling with scratch in middle school classrooms," in *SIGCSE '12*, New York, NY, USA, pp. 433–438.

[4] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari, "Learning computer science concepts with scratch," in *ICER '10*, pp. 69–76.

[5] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: Urban youth learning programming with scratch," *SIGCSE Bull.*, vol. 40, no. 1, pp. 367–371, Mar. 2008.

[6] K. Powers, S. Ecott, and L. M. Hirshfield, "Through the looking glass: Teaching CS0 with Alice," *SIGCSE Bull.*, vol. 39, no. 1, Mar. 2007.

[7] D. Parsons and P. Haden, "Programming osmosis: Knowledge transfer from imperative to visual programming environments," 2007.

[8] M. Homer and J. Noble, "A tile-based editor for a textual programming language," in *VISSOFT '13*, Sept 2013, pp. 1–4.

[9] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow, "Seeking Grace: a new object-oriented language for novices," in *SIGCSE*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445240

[10] M. Homer, T. Jones, J. Noble, K. B. Bruce, and A. P. Black, "Graceful dialects," in *ECOOP*, 2014, Preprint: http://ecs.vuw.ac.nz/~mwh/ecoop2014.pdf.

[11] C. Lewis, S. Esper, V. Bhattacharyya, N. Fa-Kaji, N. Dominguez, and A. Schlesinger, "Children's perceptions of what counts as a programming language," *J. Comput. Sci. Coll.*, vol. 29, no. 4, Apr. 2014.

[12] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. Scratch," in *SIGCSE '10*.

[13] D. B. Palumbo, "Programming language/problem-solving research: a review of relevant issues," *Review of Educational Research*, vol. 60, no. 1, pp. 65–89, 1990.

[14] R. A. Jeffries, "Comparison of debugging behavior of novice and expert programmers," in *AERA Annual Meeting*, 1982.

[15] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-user debugging strategies: A sensemaking perspective," *ACM Trans. Comput.-Hum. Interact.*, vol. 19, no. 1, May 2012.

[16] K. M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children," *ACM Trans. Comput.-Hum. Interact.*, vol. 8, no. 1, pp. 1–33, Mar. 2001.

[17] D. J. Gillan, K. Holden, S. Adam, M. Rudisill, and L. Magee, "How does Fitts' law fit pointing and dragging?" in *CHI '90*.

[18] I. S. MacKenzie, A. Sellen, and W. A. S. Buxton, "A comparison of input devices in element pointing and dragging tasks," in *CHI '91*.

[19] "Blockly," https://code.google.com/p/blockly/.

[20] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *SIGCSE Bulletin*, vol. 35, no. 1, 2003.

[21] W. Barendregt and M. M. Bekker, "Children may expect drag-and-drop instead of point-and-click," in *CHI EA '11*, 2011.