A Tile-based Editor for a Textual Programming Language

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 continuing on to traditional textual environments, all within the same programming language.**

I. Introduction

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. These environments present a limited language with a restricted expressive domain, meaning that eventually programmers must move on to a "real" textual programming language. Grace is a new object-oriented language with a conventional textual syntax, being designed to teach novices to program. 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.

In the next section we will give a brief description of the important design features of the Grace language. In Section III we describe Tiled Grace and explain the design choices we made in it. Section V positions Tiled Grace among related work, and Section VII concludes.

II. Grace

Grace [2] 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.

A key goal of the language is that students who have mastered programming in Grace should find it easy to transition to other languages. To that end, 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" [3] 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 complete program and its output is shown in Figure 1. Tiles may be dropped anywhere in the large leftmost pane, and the user can construct different sub-programs in different parts of the editing pane. Different kinds of tile are shown in different colours, with closely related concepts, such as variable declaration, reference, and assignment, having similar colouring. [1]

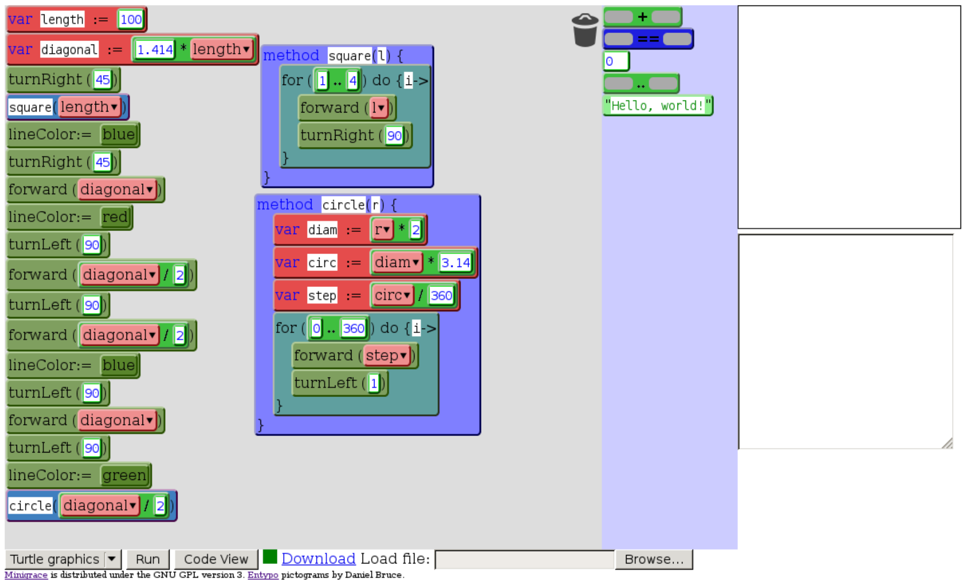
Some tiles have "holes" for other tiles to be dropped into, such as tiles representing operators or method requests. Other tiles have input fields for the user to enter a number or string constant, or to name a variable or method. Where there is a hole, the user can drag an appropriate tile from the toolbox (the narrow centre-right pane) into the marked empty space in the destination tile, which will expand to fit its new contents.

The feel of Tiled Grace is similar to Scratch [1] and Blockly [4], which inspired this work (see also Section V). Tiled Grace differs in that it is backed by a genuine textual language: the tiles themselves 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 the textual representation the user can switch to a standard syntax-highlighted view at any time.

Figure 2 shows this transition in progress: while editing the same program as shown in Figure 1, the user has switched to a textual view, and the tiled view has transitioned to syntax-highlighted code, while remaining in the same physical location. The code blocks move smoothly into place, finishing with a traditional linear textual editor view as shown in the last frame. In this way, the relationship between tiles and the corresponding part of the textual program is clearly visible. Each separate 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 across the lifetime of the program. This text is editable if the user

[1] In the present prototype, these colours are simply assigned arbitrarily in sequence around the colour wheel.

©2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
This is the author's version of this paper, published in the IEEE Working Conference on Software Visualisation (VISSOFT), 27-28 September 2013, Eindhoven, Netherlands, pp 1–4. The final publication is available in IEEE Xplore via http://dx.doi.org/10.1109/VISSOFT.2013.6650546.



```
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 square(l) {
  for (1 .. 4) do {i->
    forward (l)
    turnRight (90)
  }
}

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)
  }
}
```

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.

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, a self-hosted compiler for the Grace language [5] that targets C and JavaScript. Tiled Grace runs in a web browser without installation, and can be accessed at http://ecs.vuw.ac.nz/~mwh/minigrace/tiled/. Tiled Grace run in recent versions of Firefox, Chrome, and Internet Explorer[2], but does not work in other browsers (notably Safari and Opera) at the time of writing.

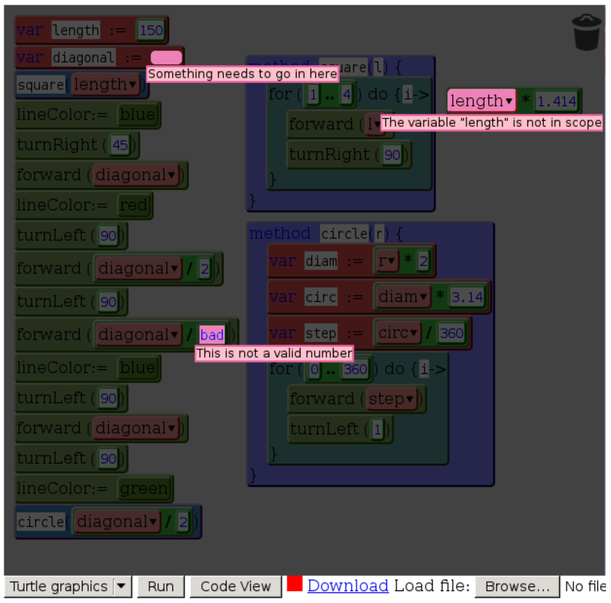
The Tiled Grace prototype interface is pictured in full in Figure 1. The code area is the large box in the top left; next to it is the toolbox of available tiles. Different sets of tiles can be selected by hovering over the toolbox with the mouse pointer and choosing a category. On the far right are the drawing and textual output areas. Below the code area are the options to run, switch views, and load or save. The user can select their dialect from the drop-down list in the bottom left, and switch views with the "Code View" button. The green indicator square next to the "Code View" button shows if the program can be compiled (green) or has errors (red). The user can download their program as ordinary Grace source code that can run on other implementations, and load source files back into the system.

IV. Functionality

A. *Handling Errors*

The very duality of view Tiled Grace is built around creates new opportunities for error. As well as the common errors of textual editing, tiles permit other forms of error, and the

[2]Because of technical limitations, the overlays on hover described in Section IV-B do not function in Internet Explorer, but some overlays are available by right-clicking on a tile.



```
var length := 150
var diagonal :=
square(length)
lineColor:= blue
turnRight (45)
forward (diagonal)
lineColor:= red
turnLeft (90)
forward (diagonal / 2)
turnLeft (90)
forward (diagonal / bad)
lineColor:= blue
turnLeft (90)
forward (diagonal)
turnLeft (90)
lineColor:= green
circle(diagonal / 2)

method square(l) {
  for (1 .. 4) do {i->
    forward (l)
    turnRight (90)
  }
}

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)
  }
}

length * 1.414
```

Something needs to go in here

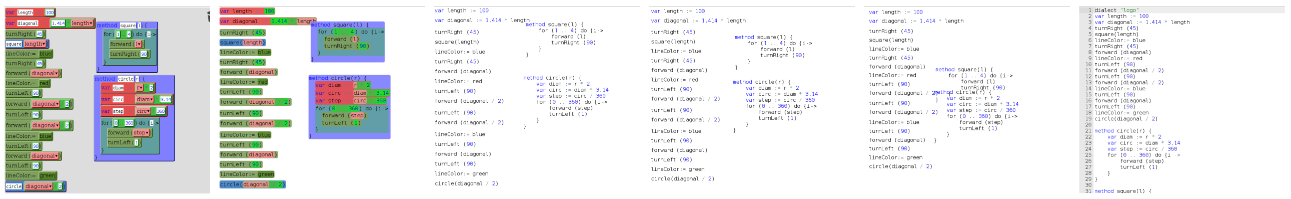
The variable "length" is not in scope

This is not a valid number

Fig. 3. Errors displayed in a modified version of the turtle graphics example.

interface between the two forms must prevent errors spreading from one to the other.

While the tiled view prevents most syntax errors, the user may still omit to fill in required components — for example, not specifying a variable name, leaving the hole on one side of an operator empty, or moving a reference to a variable outside of its scope. In each case, the textual representation of the program would be incorrect or misleading. To combat that, the user can only switch views when the program is valid: when there are unfilled holes or other errors when the user attempts to change view, the error sites will be highlighted and the view unchanged. A graphical indicator shows at all times



```
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.

whether the program is currently valid; when the indicator is red the user may hover over it to highlight all existing errors, which are labelled with their cause (for example, "Something needs to go in here" at an empty hole, or "The variable "length" is not in scope"). These error sites are shown by desaturating all of the code area except the error sites, and overlaying an associated error message at the site. An example is shown in Figure 3 where the user has hovered over the red square indicating an error, which was green for the unmodified version of the program in Figure 1. Alternative indications are possible; as well as desaturation, we have experimented with overlaid arrows, and intend to investigate the use of borders, animation, and combinations of these indications.

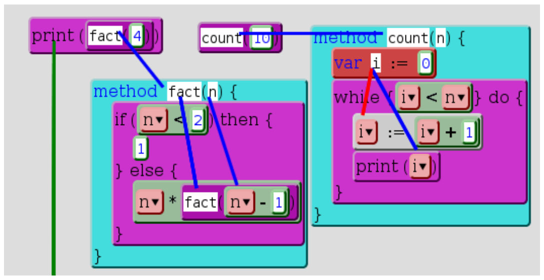
In the text view, the user is unrestricted in the kinds of error they can produce, as in any textual editor. The code is continually compiled in the background and errors marked where they occur. If the user tries to switch to the tiled view while the program does not compile, they will be presented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

By ensuring the program is valid when changing views, errors are not retained any further than necessary and no additional long-term errors are created. This was a difficult choice: some errors would be easier to solve if the user could look at the program in two ways. In future we may allow at least some classes of error to pass through the barrier between the two views, but for the moment have chosen to ensure that the program is always valid immediately after a transition.

B. Overlays

As well as visualising the code itself as tiles, Tiled Grace can visualise relationships between parts of the code (see Figure 4). 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. If applicable, multiple overlays may appear at once. These overlays are similar to those found in spreadsheets to illustrate the dependencies of a formula.

In the textual view the user may hover their mouse pointer over a variable or method to see an overlay showing the definition or use sites. In this view, the overlay is very like the similar overlay in DrRacket [6].



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

Fig. 4. Composite image of multiple overlays at once. All blue lines run between a use and the definition of a variable or method. The red line indicates a reassignment of a variable; the green indicates a method from the dialect, pointing at the dialect selector. Only one of these overlays can be shown at one time in reality.

C. Dialects

Grace dialects can extend the methods available to the programmer (as in the turtle graphics dialect in Figure 1) and restrict features of the language or create new errors [3]. Tiled Grace supports both of these components.

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. This description can be automatically generated from the dialect itself, but many dialects will benefit from manual annotations to reflect the intention of the dialect better. A simple example of when this may be desirable is when building control structures: the while()do() method takes two blocks as arguments, one as the condition (because it may be executed more than once), and one as the body of the loop. Although both parameters are blocks, the intention is different. The body is expected to contain many statements, while the condition will likely be a single brief expression. The dialect description can make this intention clear, as well as other limitations. In future, annotations within the dialect itself may specify these intentions.

Dialects are an important generalisation of the ability in Blockly to choose an extended sub-language to use. Because these dialects persist in textual form, and even originate in it, the user retains the ability to use use and understand them even outside Tiled Grace itself. Our dialects may also define and report new classes of error, shown in the same way as all other errors.

V. Related Work

Scratch [1] is a wholly visual drag-and-drop programming environment with jigsaw puzzle–style pieces, aimed at novices and children. Scratch programs manipulate a persistent microworld; the Scratch environment also includes a persistent graphical area which may contain multiple "sprites", each of which has its own independent code associated and may move, draw, or display messages from itself. Scratch takes full advantage of its purely-graphical nature; the shape of each tile maps exactly to where it is syntactically valid, and some tiles combine what would be multiple concepts in most languages into a single element. The Scratch language follows a concurrent event-driven model, where many pieces of code may be executing at once. Scratch has been found useful for motivating new programmers to begin exploring the ideas of programming, and inspired this work. Unlike Grace, Scratch code does not have a textual form and cannot be "written".

Blockly [4] is very similar in ethos to Scratch, but incorporates multiple variant languages which can be extended with JavaScript code, and lacks the persistent world of Scratch. Blockly runs entirely in a web browser. The user can export their Blockly program to JavaScript or Python, but there is no editable textual format.

Alice [7] is a 3D microworld language manipulated by drag-and-drop. The Alice IDE allows users to drop 3D models into the world and associate logic with them. Each object in the world is also an object in the sense of object orientation, and can respond to events and messages from outside. All code editing is by drag-and-drop; there is no concrete syntax, although recent versions of Alice can also export code to Java.

Greenfoot [8] is an IDE for a subset of Java, presenting a graphical microworld based on the Actor model. Users write textual source code, but many high-level concepts are available as built-in methods of the world or of all actors. Code is written and accessed only with textual Java syntax, which users must learn assisted by common IDE features.

DrRacket [6] is an IDE for the Racket language, a dialect of Scheme aimed at education. The editor is purely textual, but includes an overlay system linking definitions with their usages when the mouse is hovered over a term in the editor, as described for Tiled Grace in Section IV-B. DrRacket does not include any alternative representations of a program, nor attempt to visualise the editing in any other way.

TouchDevelop [9] integrates an essentially textual language with an IDE aimed at touch-screen usage. The IDE avoids most use of textual input by having the user manipulate the syntax tree itself: the user touches where they want to change and the IDE presents them with a list of options they can put there, or will prompt them to populate required areas of new code they add. The syntax is reasonably conventional, although symbols are used to mark method calls and some aspects, such as comments, are shown only by typographic features. The interface is designed to be used on tablets and mobile phones, as are the resulting programs, but they may also be used without a touch screen. Programs are always shown textually with light visual annotation, and editing always corresponds essentially to a textual insertion or deletion.

VI. Future work

Currently, Tiled Grace does not consider static type information at all. We intend to consider adding at least basic type checking for simple cases — for example, trying to subtract strings — to avoid mistakes that users might make more easily with the tiled interface. We would also like, if possible, to signal these errors and others 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 not complete; 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 needs further work and consideration. The current colouring of tiles is essentially arbitrary, while the overlays are functional but may sometimes obscure important 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.

Finally, we plan to conduct empirical evaluations of Tiled Grace, focusing particularly on how (and if) Tiled Grace's multiple views and animations help novices move from graphical to textual syntax.

VII. Conclusion

Tiled Grace is a graphical editing environment for Grace, a new object-oriented programming language aimed at education. 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 hope that Tiled Grace can ease the barrier of entry into programming for novices while avoiding the need to re-learn programming concepts when moving on to a textual language.

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] 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.

[3] M. Homer, J. Noble, K. B. Bruce, and A. P. Black, "Modules and dialects as objects in Grace," School of Engineering and Computer Science, Victoria University of Wellington, Tech. Rep. ECSTR13-02, Mar. 2013, `http://ecs.victoria.ac.nz/Main/Technical-ReportSeries`.

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

[5] A. P. Black, K. B. Bruce, M. Homer, and J. Noble, "Grace: the absence of (inessential) difficulty," in *Onward!*, 2012, pp. 85–98.

[6] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *PLDI*, 2011.

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

[8] M. Kölling, "The Greenfoot programming environment," *ACM Transactions On Computer Education*, vol. 10, no. 4, p. 14, 2010.

[9] R. N. Horspool, J. Bishop, A. Samuel, N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich, *TouchDevelop: Programming on the Go*. Microsoft Research, 2013.