Lessons in Combining Block-based and Textual Programming

Michael Homer and James Noble
Victoria University of Wellington
New Zealand
{mwh,kjx}@ecs.vuw.ac.nz

Abstract *Tiled Grace is a block-based programming system backed by a conventional textual language that allows switching back and forth between block-based and textual editing of the same code at any time. We discuss the design choices of Tiled Grace in light of existing research and a user experiment conducted with it. We also examine the sorts of task preferred in each mode by users who had the choice of editing either as blocks or as text, and find both positive and cautionary notes for block-based programming in the results.*

1. Introduction

With Tiled Grace, we aimed to produce a block-based programming system that was fully integrated with a conventional textual language aimed at education, Grace [1–3], to allow learners to make the transition from blocks to text gradually over time. At any time, a programmer can with the click of a button switch from editing their code as blocks to editing it as text, and back again, as often as desired and for as long as required. Tiled Grace matches its block syntax exactly with the syntax of the textual language to reinforce knowledge of both, and uses animations to show the correspondence between the two representations while transitioning. Like the textual Grace language, Tiled Grace aims to be a guiding step for novices who will move on to other languages or paradigms.

In this paper we explore our specific design choices made while building Tiled Grace and elaborate on their motivation in relation to existing work in both visual languages and educational psychology. We reflect on those choices in retrospect in light of both subsequent literature and a usability experiment we conducted with the tool. This experiment aimed to determine:

- whether this ability to switch views would actually be used, or if users would merely use one or the other;
- whether the novel animated transition connecting the two views was appreciated, or found confusing or unhelpful;
- whether the error reporting system we had created for the tiled view was helpful, as it was entirely experimental;
- how engaged users were with the tool, as a system that users do not enjoy will not be used;
- and any unanticipated difficulties or usage.

DOI reference number: 10.18293/VLSS2017-007

Our experiment with Tiled Grace also offers a unique opportunity for analysis. For the first time, programmers had the opportunity to edit the same program as both blocks and as text, and particularly to edit *parts* of the program in each mode. We perform a new analysis of the dataset in this paper, examining the choices and revealed preferences of the users in the experiment, with a further goal of finding:

- which tasks users preferred to perform as text and which as tiles;
- and how these patterns vary by the experience level of the user.

From all of this we attempt to draw lessons for the future of block-based programming. While we see a number of positive signs for these languages and editors, we also find some cautionary notes where enthusiasm may not match reality.

In the next section we briefly introduce Tiled Grace at a high level. After that, we outline the more relevant similar systems, and then discuss Tiled Grace's design choices for usability and learnability in relation to others, and where we made trade-offs to support our goal of integrating text and blocks. We then summarise a past usability experiment with the tool, and go on to present a novel analysis of the actual use of each modality, before relating and comparing our results and experience with what has been reported by others. Finally, we attempt to draw some lessons for block languages from both our experience in building a hybrid system and our experimental results.

2. Tiled Grace

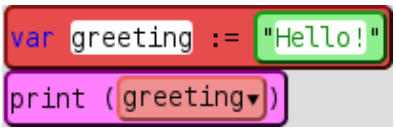
We will briefly introduce the Grace language generally: Grace is a textual, object-oriented, block-structured, curly-brace language aimed at novice programmers, primarily in tertiary study [1]. It aims to support new programmers in the first year or two so that they can develop the understanding required to learn new languages for their later careers, and includes a number of design choices intended to assist that. In this work, we are building on Grace, and on an implementation of Grace, leveraging the existing pedagogical design and implementation work already carried out on the textual language.

Tiled Grace [4, 5] presents an editing environment for Grace programs based on drag-and-drop *tiles*. The basic structure

is common across drag-and-drop block-based programming systems. The tiles for a string and variable, for example, appear as:

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 to be assigned:

The holes are the empty grey rounded-rectangular areas. The user can drag a tile inside a hole to build up their program. 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.



A complete program and its output in progress 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 selected from a pop-up menu that appears when using the toolbox. At the bottom of Figure 1 the dialect selector, run button, and other interface controls are displayed.

The user can switch to a textual editor at any time. The transition from tiled to textual view is shown through a smooth animation where each tile and block of code has a continuous visual identity throughout the transition.

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 chooses to switch back to tiles, the same behaviour occurs in reverse.

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. First the tiles fade out to blocks of the corresponding syntax-highlighted textual code, while remaining in the same physical location (frame (b)). The code blocks then glide into place (frame (c)), finishing in a linear textual ordering. Finally, the tiles become fully editable ordinary text, as shown in frame (d). 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. The ordering between them in the textual display is arbitrary, but consistent across the lifetime of the program. The displayed 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. When exported, a meaningful comment is appended to the end of the program stating the coordinates of each independent "chunk" in pixels, while the chunks are separated from each other by blank lines; within the system the location information is stored in memory during a text-editing session. If a new chunk of code is added, it is assigned a default location upon switching back to tiles.

Tiled Grace innately supports *dialects*, a language variation feature of the textual Grace language [6]. Dialects can provide new features to the language and new tiles in the toolbox, and impose additional restrictions on what can be written. Each module [7] of the program can use a different dialect. Different dialects can provide drastically different sublanguages, with their own tiles and rules, within the same overall syntax and semantics. Because all Grace control structures are defined as methods to begin with, a dialect can introduce its own control structures at will and they will fit in with the rest of the language.

Tiled Grace runs in a commodity web browser, including both the editor and the backing compiler. It can be accessed at http://ecs.vuw.ac.nz/~mwh/minigrace/tiled/, and works at least in recent versions of Firefox, Chrome, and Internet Explorer/Edge. Tiled code is converted behind the scenes to textual Grace code, which is then compiled into JavaScript for execution. The compiler provides a syntax-tree export that is used to transform textual code back into tiles, or to import new textual code.

We will discuss other features of Tiled Grace in relation to the motivations that inspired them in the next section.

3. Existing Block Programming Systems

Several block-based programming languages and systems are in current use, the most well-known of which is probably Scratch [8]. We will briefly introduce the systems most relevant to Tiled Grace, focusing on the aspects that relate to this work. The design dimensions we consider interesting are:

- the role of a **textual modality**: is there none available, export to another language, export to and import from another language, switchable views, or simultaneous display?
- the sort of **block positioning** that it allows: freeform layout or a fixed structure.
- **when errors** are reported: is it when they are introduced, when starting to run the program, at runtime, or never?
- **how errors** are reported: in-place, in a list, one at a time in a fixed place, or with a marker. (In both of these dimensions we are interested in syntax, structure, and type errors, rather than logical errors — the kinds of error that a reasonably conventional textual language might report statically).

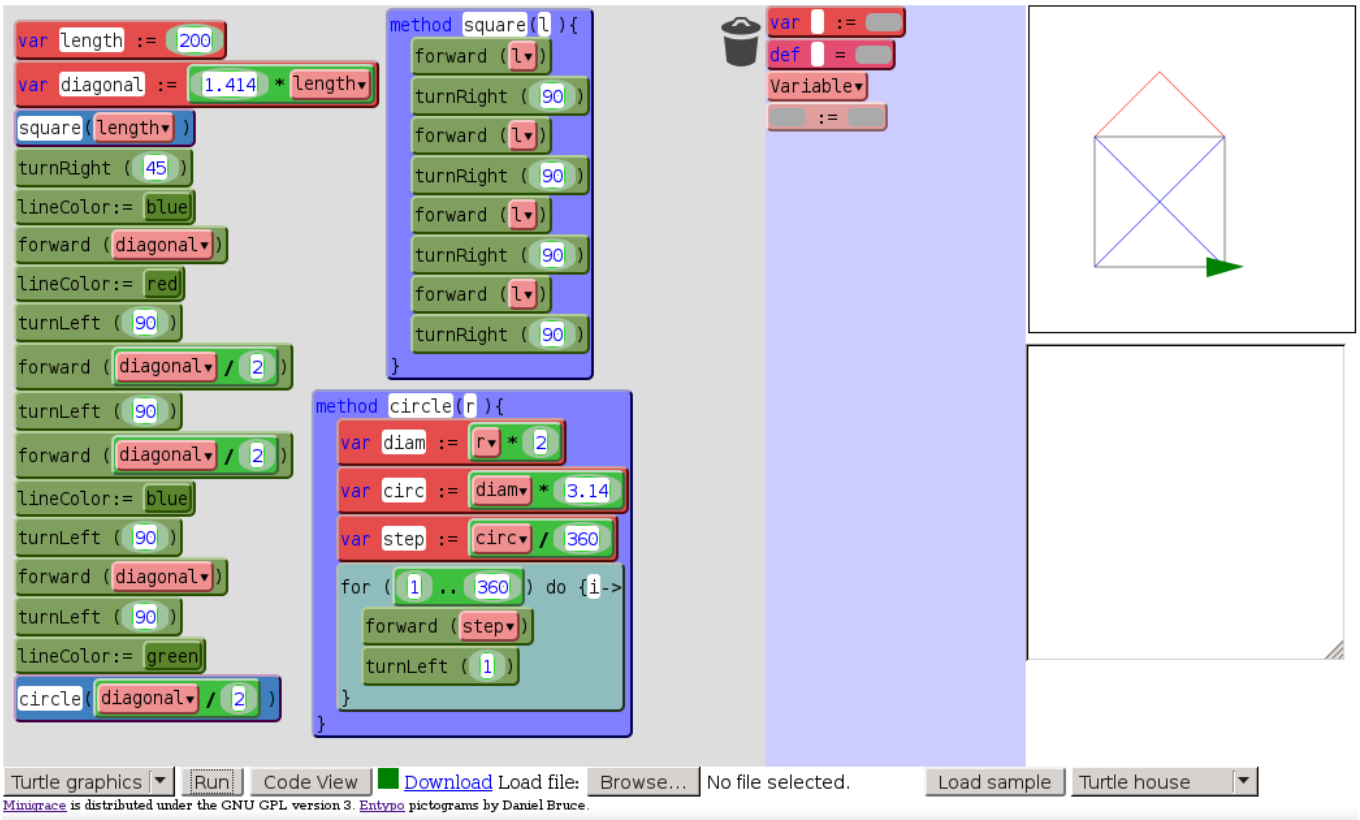


Figure 1. Tiled Grace editing a small program in the "turtle graphics" dialect, currently executing

- are different **types distinguished** by shape, colour, repulsion from incompatible locations, not at all, or some other means?
- how are **dependencies** between different parts of the code (for example, variables and their references) maintained or indicated: not at all, with an overlay, with automated scoped renaming, or something else?

Table 1 summarises each system we consider against each of these dimensions. Tiled Grace has, respectively, a switchable text view, freeform layout, errors shown both when introduced and when run, errors displayed in place, repulsion of incompatible types, and an overlay showing uses and definitions of variables and methods along with automated renaming.

3.1. Scratch

The visual side of Tiled Grace is most similar to Scratch [8], 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 (although its blocks are all dependent on textual labels). A key boon of Scratch is its immediate graphical microworld. A student can instantly see the effects of running a piece of code, live in front of them. Code can be modified during execution with instant feedback.

Scratch has been very successful in driving engagement, particularly with children. New users who might never have considered programming a computer take to it quickly and begin exploratory programming with little prompting. We observed this engagement ourselves while working with Scratch in an outreach programme to a local school, which led to the conception of Tiled Grace as a way to gain this engagement within a more complete language. Scratch has also proven useful for a variety of other purposes, including driving social interaction between children, promoting storytelling, and teaching music; we did not focus on these areas for Tiled Grace and will not address them further in this paper.

Scratch has freeform positioning of blocks anywhere in the workspace, and uses different block shapes to distinguish types of value. Scratch does not regard any program as erroneous, and will attempt to execute any program, skipping over missing or invalid parts. There is no innate textual form of Scratch, though a debugging export of an entire program is available by a hidden option.

3.2. Squeak Etoys

Etoys is a tile-based programming system built on the Squeak Smalltalk system [9]. Etoys focuses on exploratory learning in general. Textual code equivalent, but not identical, to the tiled code can be exported and executed. Tiled code is always valid, with type errors prevented from being constructed, while erroneous textual code cannot be activated. Tiles corresponding to abilities of each item in the world are

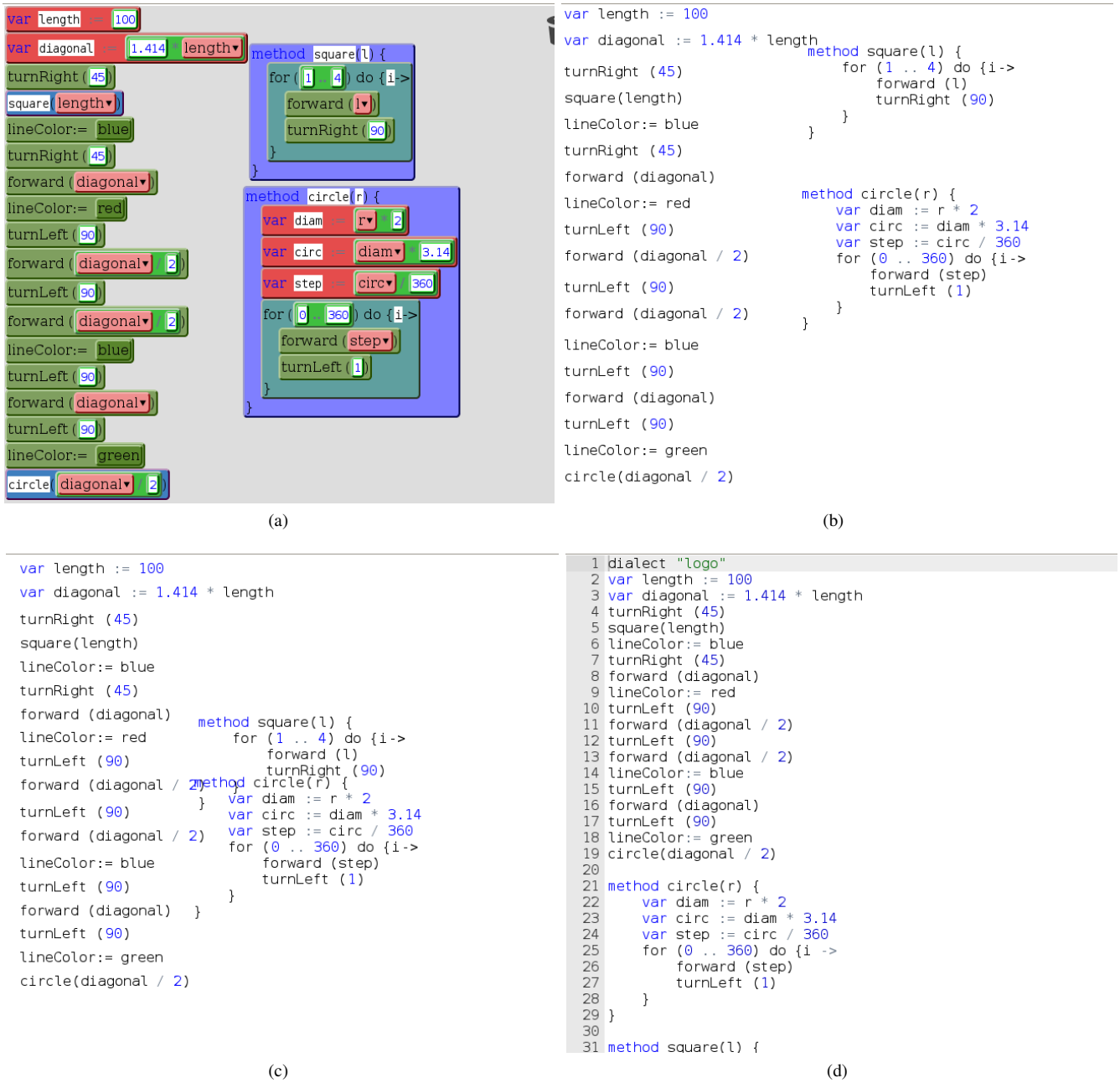


Figure 2. Frames of the animated transition from tiled to textual view. (a) Tiled. (b) Fade backgrounds and highlight syntax. (c) Glide tiles towards their positions in the text. (d) Switch to an actual text editor in-place. Transitioning from textual to tiled view shows the same intermediate states in reverse. The transition from tiles to code, and the movement of code, is smoothly animated.

accessible through a menu on that item, rather than a general broad menu. An interesting aspect of Etoys is that code tiles exist within the same world as user objects, and tiles can be created representing the physical display of blocks of code. These tiles can be used like any other to manipulate the display of the code.

3.3. Alice

Alice [10] is similarly microworld-driven, but aimed at a slightly older audience. Alice has objects in the three-dimensional microworld that are also objects in the object-orientation sense, with a primarily event-driven programming model with common structured-programming features as well. Alice makes heavier use of menus than Scratch, but also has a higher degree of enforced structure. These menus provide on-demand exposure of relevant possibilities and encourage a different style of experimental programming than Scratch's toolbox.

Alice has structured positioning of blocks, with (subtle) shape indicators for some types. Alice supports exporting to

Table 1. Many existing block programming systems laid out according to their design elements relating to each design dimension.

Language	Text	Positioning	When errors	Where errors	Types	Dependencies
Scratch	None	Freeform	Never	n/a	Shape	Renaming (some)
Etoys	Export	Freeform	Never	n/a	Repulsion	Renaming
Alice	Export	Structured	Intro., start	One, list	Shape	Renaming
Blockly	Export	Freeform	Start	Varies	Repulsion	None
App Inventor	None	Freeform	Intro., start	Marker	Repulsion	Renaming
Pencil Code	Switchable	Structured	Start	One	No	None
BlockEditor	Export/Import	Structured	Start	One, list	Shape (some)	None
GP	Export/Import	Freeform	Never	n/a	Shape	Renaming
Calico Jigsaw	Export	Freeform	Runtime	One	No	None
TouchDevelop	None	Structured	Intro.	One	Repulsion	Renaming
Greenfoot	Read-only	Structured	Intro.	Marker	n/a	None
Tiled Grace	Switchable	Freeform	Intro., start	In-place	Repulsion	Overlay, renaming

Java and reports many errors upon introduction, and some when starting the program.

3.4. Blockly

Blockly [11] is very similar in ethos to Scratch, with freeform positioning. Blockly runs in a web browser and incorporates language variants (what we call *dialects*), but in mimicking Scratch also has no editable textual format. Blockly's goal is to support developers embedding a visual language into other systems, both educational and otherwise.

Blockly supports exporting code to a number of languages, but these exports are not bijective. There is no explicit indication of which parts of the visual representation correspond to which parts of the textual representation. Because Blockly is designed for embedding, a range of different behaviours can be provided by the embedder, but repulsion from invalid locations is built in.

3.5. App Inventor

App Inventor [12, 13] is one of a number of Blockly clients, aimed at teaching. App Inventor was extended with a textual language, TAIL, semantically equivalent to its block language [14, 15]. An aspect of the TAIL integration that most systems do not match is the ability to embed a portion of textual code within the block view, as a block containing the text. This feature of TAIL is one that would be particularly useful in systems when there are parts of the textual language that the blocks cannot express, but came after our work so we did not include it.

As well as TAIL, a Python export for App Inventor has been proposed [16]. Neither is currently in the production release. App Inventor marks erroneous blocks with a persistent indicator when the error is detected, and the user can inspect the block individually for an error report. It has no tracking of interdependencies beyond renaming variables.

3.6. Droplet and Pencil Code

Droplet [17, 18] and the closely-related Pencil Code [19] slightly postdate the genesis of Tiled Grace, but also attempt to bridge blocks and text. Pencil Code concentrates on straight-line programs in a Logo-like turtle graphics system and simple audio/drawing programs, and supports editing large subsets of both CoffeeScript and JavaScript as blocks and text, with both block and text editing for each language.

Droplet introduced an animated transition that parallels Tiled Grace's and is now part of mainline Pencil Code with many users. Droplet is a general library supporting any language with an appropriate adapter, but the main use is in Pencil Code with CoffeeScript. Text is treated as the primary representation in Droplet, and it retains complete or nearly-complete information from the source, including comments and layout (which Tiled Grace does not only due to technical limitations of the underlying compiler). Were we creating Tiled Grace now, we would likely build on the Droplet library instead of a bespoke system.

Pencil Code reports some errors upon trying to run the program by way of a popup. Types are not indicated in any way. Blocks are positioned in a structured fashion, and no dependencies are tracked. Switching between block and textual view is possible at any time, except when textual syntax errors exist.

3.7. BlockEditor

Matsuzawa *et al* [20] built BlockEditor, an editor for a visual language called Block that can save the program to textual Java. The Java code can be edited and automatically reimported into BlockEditor. In this way a learner can move between the two languages at will, continuing with the same code. The Block visual language is not exactly the same as the Java textual language, but parallels the structure closely enough for a bijection to exist for the programs in an introductory course. An experiment over a first programming course for non-majors found that users did use both modes, with the rate of Block use trending downwards and Java upwards as a course progressed, and that higher usage of the visual mode corresponded with lower self-efficacy.

BlockEditor allows exporting to Java, editing, and re-importing the code. The block and textual code looks dis-

similar, but is semantically equivalent. Shape is used to mark some types of sockets.

3.8. GP

GP is a general-purpose blocks programming language intending to be useful for casual programmers other than children [21]. GP is principally block-based (building on Snap! [22]), but experimentally has had an editable text mode (exposing the underlying LISP-like language), highly condensed blocks appearing as text, and text-based insertion of blocks. The text-like blocks were found to be less jarring (though less powerful) than exposing export and import of the underlying data structure had been. Permitting not only animation but the ability to select intermediate points exposes the text-block transition to a further degree than other systems, including Tiled Grace.

3.9. Calico Jigsaw

Calico [23] is a multi-language IDE for introductory programming, which includes a visual language called Jigsaw. Jigsaw uses puzzle pieces and drag-and-drop, and the Calico system enables exporting the program to other textual languages, primarily Python. The Jigsaw syntax is distinct from any textual language and export is to a text file. Jigsaw allows a degree of freeform positioning of blocks, and reports errors primarily at run time, with the triggering block marked.

3.10. TouchDevelop

TouchDevelop [24] integrates an essentially textual language with an IDE aimed at touch-screen usage, rendering the program as large blocks. 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. When the programmer adds a new element the system will prompt them to fill in any required arguments, like the condition of a loop. While the syntax is reasonably conventional, there is no direct textual form of TouchDevelop code, and some aspects, such as comments, are shown only by typographical features. Editing always corresponds essentially to textual insertion or deletion. TouchDevelop has fully structured positioning and enforces that the program is well-formed whenever possible.

3.11. Greenfoot and Stride

Greenfoot is an introductory programming system with a two-dimensional microworld, which has recently been extended with a "frame-based" editor [25, 26]. Like a block system, Greenfoot's frame-based editor presents hierarchically-related elements of the code as nested indivisible oblongs with slots for subsidiary elements, but like a textual language interaction and input is principally with the keyboard. Its Stride language reuses the concepts of Java and uses textual labels and structure closely matching Java syntax.

The programmer manipulates the syntax tree at the level of individual nodes using single-letter keyboard shortcuts, but some slots use free text entry even for structured elements (such as method references or loop conditions), so Stride is a hybrid structured and unstructured editor. These unstructured fields are the only place that syntactic errors can be introduced, other than empty mandatory fields. An individual module is always a well-formed class down to some subsidiary point.

A Java view of the module can be shown at any time, including while the program is in an erroneous state. This view is not editable, but inserts the necessary braces, comment markers, and other syntactic structure, while removing additional labels present in the frame view, with an animated transition preserving the identity of the elements common to both views.

3.12. Other systems

A number of other systems, or experimental systems, have incorporated some level of textual code alongside blocks. These include a simultaneous-display version of Snap! [27] showing JavaScript code (not part of mainline Snap! development), and systems for defining extension blocks using some host textual language [22, 28, 29]. We discuss the Snap! extension briefly below, but consider merely-extensible systems out of scope for this paper.

4. Designing Tiled Grace

Why build Tiled Grace when Scratch, Alice, and similar systems already exist? Our design goal for Tiled Grace was to provide the engagement and lessened syntactic burden of these existing systems, while introducing the concepts of a textual language at the same time so that the user could transition into it at their own pace, in accordance with educational psychology principles. We built on top of an existing conventional textual language aimed at education in order to leverage existing education design work, rather than reinventing it, building an interface for editing that textual code rather than a new language. In this section we break down some motivating aspects of our design, particularly in relation to other languages and approaches we built on or steered away from.

4.1. Migration

A key goal in Tiled Grace was to enable its own obsolescence for each user in their own time, where they could move on to the textual paradigm when ready and with the support to do so successfully. Building on a general-purpose language ensured that there was no functional limitation in the tiled view (as contrasted with microworld-focused languages), but did not alone ensure that it would be feasible to move on. Ultimately, Tiled Grace aimed to ease beginning with Grace, to match the Grace language's goal of easing beginning with programming, in both cases as an initial step only.

A well-reported problem with moving on from visual to textual languages [30], 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. In particular,

it is known from both educational psychology in general, and computer science education specifically, that transitioning between languages early in learning is unhelpful [31], or indeed any attempted *transfer of learning* at an early stage without very careful structuring [32, 33].

For learners to achieve transfer they must be taught the concepts in a fashion that facilitates transfer [32]. Without such teaching the knowledge tends to be *inert*: it can be applied within its original context, but learners will not generalise from that context to apply their knowledge elsewhere. Perkins and Martin found that students learning to program would learn language constructs inertly, and so had difficulty applying their knowledge to the act of programming, notwithstanding that the distance of transfer is minimal in this case [34], while Dyck and Mayer found that without transfer-focused teaching learners of BASIC would master the syntax of the language, but struggle more with semantics than those taught with transfer [35]. An assumption in much teaching is that transfer to similar domains will occur automatically, but research has not borne this assumption out in practice [32], instead finding that instruction must be tailored to assist transfer; in the literature, this tailoring to target explicit transfer is called *bridging* [35–37]

These ideas and experiences were a strong influence on our design of Tiled Grace. In particular, the animated transition between visual and textual representation aims to assist bridging by demonstrating the exact parallel between the two sides. Similarly, we made the block structure match the syntax of the textual language, and even display the relevant syntax in-place on each tile. In this way the user was always seeing the textual syntax, even while editing blocks, and would gain some familiarity with what to expect in text. Using text as the primary representation also ensured a convenient interchange format for both whole and partial programs.

Restricting ourselves to an exact match with the textual syntax limited what we could do in the blocks. Unlike other visual block-based languages we could not use additional layout or components within a single tile to make the block language simpler (for example, using multiple successive holes without intervening syntax, adding extra labels, or physically offsetting or aligning some fields to distinguish them with no other syntax), because that would break the direct correspondence with the textual syntax we did not control. This is one area where our goal of integrating both worlds has made the system weaker in respect of one approach or the other than a "pure" block or text language.

Given all of the above, there is a fair question in the air — why switch to textual languages at all? Aside from the dearth of professional block languages for those students who would like a job in future, a key issue in existing block languages is that using them is exhausting, having high "viscosity" [13, 38] — the difficulty of making a local change. Even reading and understanding a complex program with many nested blocks can be difficult [14]. While designing Tiled Grace we knew from our own experience that as we had come to know the system better we had found the drag-and-drop interface of Scratch increasingly tiresome to use. Moving to the toolbox, dragging a tile out, switching to a different pane, finding the next tile, and so on, becomes repetitive and frustrating over time. Novice users, however, do not find this: when everything is new, the impact of retrieving each tile is unnoticed next to the difficulty of the concepts being dealt with. The toolbox is an excellent discovery mechanism for novices, the ease of getting something going is a significant driver of engagement, and the lack of syntax errors removes a major confound faced by novices. Novices are eventually no longer quite so novice, and so may want to move on.

A commonly-repeated maxim is that a good programmer can easily learn a new language. Novices are not good programmers, however, and a course structure predicated on making a language transition will likely run into trouble. Nonetheless, introductory tertiary courses in Scratch and Alice move on to other languages early, often within the first course, as programs become too complex for such languages. This has implications for the design of educational languages more generally as well: because an educational language explicitly expects learners to move on to other languages afterwards, the language must support the learner for long enough to allow them to build sufficient competence that they can successfully transfer their skills to another language.

4.2. Event versus Process

One issue with language transitions is that they are essentially "one-way" *events*: the learner must apply what they know about the earlier language to the later, but movement in the other direction is restricted. This is a problem not only for transferring concepts, but because this "event" model makes the two sides seem qualitatively different and opposed.

Powers, Ecott, and Hirshfield found that students learning Alice and a textual language in the same course frequently felt that Alice was not a "real" language [30]. Students who struggled with the textual-language part of the course felt that what they had been doing in Alice "didn't count" or was "too easy", that textual code was "real programming" and were inclined towards believing that they were not actually capable of programming; this inclination is harmful in itself. Lewis *et al* found that more students rated a picture of random green-on-black symbols from the film *The Matrix* as "definitely" or "somewhat like" programming than an image of the Lego Mindstorms programming environment (a colourful drag-and-drop system), despite the fact that those students had been learning Scratch [39, 40]. These examples are just some of the motivating concerns we had about visual-textual transitions when setting out to design Tiled Grace.

By contrast with approaches moving between multiple languages supporting a single paradigm each, Tiled Grace has a deliberately permeable barrier: a user can use the visual language, the textual language, and the visual language again, even within the same program if desired. As in BlockEditor [20], permitting both views avoids the transition event altogether, so that moving from blocks to text becomes a *process* rather than an event. A programmer can start to move

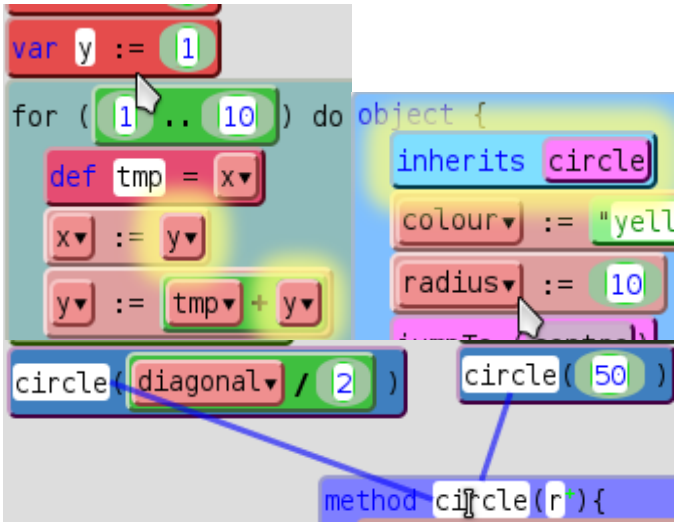


Figure 3. Overlaid dependency indicators for variables, methods, and inherited fields.

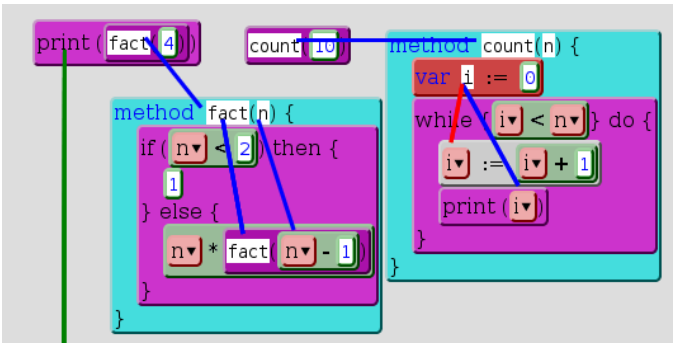


Figure 4. Composite image of multiple overlays at once in an alternative design.

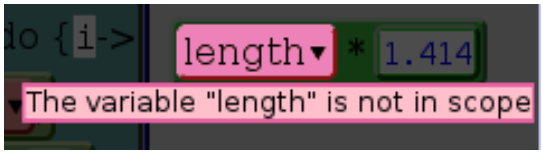


Figure 5. Marking a static scope error in the program where a variable reference has been moved out of its defined context.

to text as early as the first day, and draw out the process as long as necessary until they are truly comfortable working with textual code.

Throughout the process, both modes, and all of their past programs, remain available to the programmer. At all times, the user can see that what they were doing with blocks is exactly the same as what they are doing with text (or, indeed, the other way around). Both modalities clearly "count" as much as the other, and they are clearly both programming. Allowing movement in both directions necessitated some further trade-offs (particularly that the programmer can only switch views when there are no static errors in their program), but we considered it appropriate to the goal of the language. This textual-tiled combination was our original grounding conception for Tiled Grace.

4.3. Relationships and Dependencies

In a block language, and particularly one with arbitrary layout like Tiled Grace, it is possible for the declarations and uses of variables and methods to be dispersed around the screen where they may not be obvious, which could lead the user to break their program through being unaware of the dependencies between parts of the code. To preëmpt these incidences, we included two overlays to show the dependencies between these items.

The top left of Figure 3 shows the mouse pointer hovering over a variable declaration, with two uses of that variable highlighted. Similarly, hovering over a variable use site marks the declaration site. These indications occur anywhere in the program, for variables, constants, and method parameters. The top right shows that the "radius" field has been inherited from "circle" through a similar highlighting (the inheritance system of the underlying Grace language is object-based and blurs fields and variables [41]).

The bottom of the figure shows an unrelated method declaration, `circle`, with the mouse hovering over it. The overlay draws a line between the declaration and each use site of the method, wherever it is in the program. Again, if the user hovers the cursor over a call site, the line will indicate the declaration.

These indications are especially important to aid new users unfamiliar with the language or libraries. In our experiment, we would have people use the system with minimal training, and it was valuable that they should know the relationships between different parts of the task programs they were given. We considered an alternative dependency indicator, in Figure 4, that used lines for every indicator and showed all applicable dependencies at once. We found this overlay too busy and shifted to using the highlights from Figure 3 for everything but methods, and stopped showing possible assignment sites of variables altogether. We are not aware of a block language that successfully shows all of these dependencies, but we were inspired by the DrRacket editor for the textual Racket language [42] in the alternative approach.

4.4. Errors

While Tiled Grace was mostly modelled after Scratch, a key difference between Tiled Grace and Scratch, but much less so between, say, Tiled Grace and Alice, is that we made especial effort to provide error detection and reporting in the visual editor [5]. While in many block languages all programs can run regardless of missing or broken parts, we require that the program be well-formed to allow it to run, and prevent a wide range of errors from entering the program in the first place. To a large extent this choice is forced upon us by the need to match with a textual language, but we also believe that reporting errors early and often is beneficial.

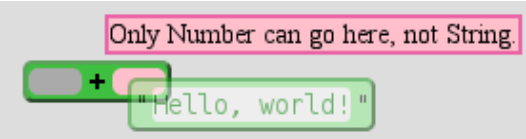


Figure 6. The display of a simple type error the user is making, where they try to place a string tile somewhere that only numbers are permitted.

While block-based editing prevents most syntax errors, the user may still omit filling in required components — for example, not specifying a variable name or leaving the hole on one side of an operator empty — or invalidate the program in other ways by moving a reference to a variable outside its scope or filling in an unsuitable value. We included a persistent graphical indicator of the current validity of the program in the interface: when it turns red, the program is somehow invalid. The user may hover over it to highlight all existing errors, which are labelled in situ with their cause (for example, an empty hole may have the message "Something needs to go in here"). 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, as seen in Figure 5. We investigated a number of approaches to indicating static errors, including overlaid arrows, persistent adjacent markers as used in App Inventor 2 [13], and a visible list of errors, but settled on this approach as a balance of space usage and clarity. The locality of error reporting and error messages is commonly better than is achieved in even advanced IDEs for conventional textual languages, which we feel is a key advantage that block-based languages can have.

Similarly, while type checking is commonly recognised as helpful in textual languages, translating that across to visual languages is challenging. We wanted to catch errors as early as possible, so it was important that we be able to detect and report type errors where possible, in a way that the user would see and understand.

We chose to use a variant on our error overlay approach to report errors as the user tries to perform the action that would cause an error, while also preventing the user from doing so. Any hole can be annotated (in the implementation) with the types it will accept. Any tile can be similarly annotated with the type of the object it represents. When the two do not match, the tile placement is not permitted. Most block languages have some variation of this "repulsion" behaviour for at least some of their tiles, but we hope that the simultaneous overlay makes clear why the tile cannot be placed where desired.

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 6, the hole is marked in pink and an error message displayed nearby: the user will not be able to drop the tile into the hole. Type errors are in this way prevented from being introduced into the program in the first place, but the user also understands why they were not able to do what they wanted. This sort of live feedback is another aspect of block languages that is both clearly useful and difficult to replicate in conventional textual languages.

4.5. Shapes

We thought it would be helpful to indicate to users the appropriate placement of tiles before they move them. Scratch partially achieves this effect through its "jigsaw puzzle" pieces: holes and tiles of different types and roles have different physical shapes, so a boolean constant or expression will not fit into a numeric expression. While immediately understandable, the approach has flaws, notably that there is a limited range of sensible shapes that can be readily distinguished and consequential limit on the number of types that can be in the system. As well, "multi-type" holes are very difficult: in Scratch it is not possible to have an array of booleans, only of its combined string-number type. These constrictions make this approach problematic to implement in Grace, as it is a language with many extensible mostly-structural [43] types, and several places that can hold variables of any type (for example, variable declarations and equality tests).

Scratch uses shapes for both types and some grammatical categories, as do a number of other block languages: some blocks can only appear at the start of a stack (rounded top), some are statements (notched), and some are values (rounded). Tiled Grace does not do this: given the underlying language, every node can be legitimately adjacent to any other, and the only node that can never appear in any kind of value position is a method declaration, while variable declarations can appear in method bodies (which are expressions) but not some other expression sites. For most nodes, it is their (return) type that would limit their possible locations, and this would provide the same effect as grammatical restrictions for these nodes. The text-as-primary philosophy of Tiled Grace means that a wide range of possible programs must be representable, particularly given that dialects may extend or replace even basic control structures.

We considered colour-coding types, such that our any-type holes would be a neutral colour, while strings, numbers, booleans, other objects, and dialect definitions would have their own colours which could be matched on both tile and hole. Similarly to using shapes, however, the number of readily-distinguishable colours is a limit on the number of types that can exist, particularly if user-defined types (such as those of custom objects or classes) are possible.

Other block-based systems not aimed at conventional programming, such as Lerner *et al*'s Polymorphic Blocks system [44] and Vasek's TypeBlocks [45], face some similar obstacles. Polymorphic Blocks uses different shapes to represent different kinds of entity, but its equivalent of our holes, "ports", behave differently. A generic, untyped port is rectangular, but it may have connected ports elsewhere, each set of which is highlighted in the same colour. When a shaped item is connected to one of these ports, the matching ports all take on the same shape. Similarly to our animated transitions, Polymorphic Blocks animates each new shape moving from

source to destination. TypeBlocks has a similar philosophy with different shapes and entities.

Polymorphic Blocks supports generic parametric polymorphism in this way, but does not (yet) address the proliferation of shapes. Within the user experiment Lerner *et al* conducted, complex shapes are created only by nesting the small number of base shapes inside one another, with scaling down when required. Conceivably some version of this scaling can be applied more broadly, but we find it difficult to imagine applying it to complex object types. Designing Tiled Grace, and observing other work subsequently, has led us to the view that block languages must make a (fairly early) choice: use jigsaw-puzzle shapes and have a very restricted number of types, or use some sort of feedback during an attempted error but allow unbounded type construction. Neither approach is innately better or worse in general, but depends on the intended application domain of the language. Nonetheless, it is a degree of fragmentation of approach that we would have preferred to avoid.

5. Experiment

We ran a usability experiment trialling Tiled Grace with 33 participants [5]. In this section we describe the procedure of the experiment and summarise the results that are relevant to this paper. An anonymised dataset from the experiment was published [46], and in Section 6 we perform a new analysis of published instrumentation data from the experiment to examine how participants used the different editing modalities.

Participants were primarily students enrolled in early undergraduate Java courses in the School of Engineering and Computer Science at Victoria University of Wellington, selected so that they would have some existing familiarity with the idea of programming. This experiment focussed on usability and engagement, rather than learning, so true novices were not considered suitable subjects at this stage. Studying learning would require pedagogical studies of textual Grace to have been completed already in order to distinguish the effect of Tiled Grace, and we were mostly interested in whether the system design was practical at this point. The 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 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.

The experiment took place in March–April 2014. Participants were asked to use Tiled Grace to write, modify, correct, and describe programs, while we measured their use of different features of the system. Participants also completed questionnaires about themselves and their use of the system. This experiment was approved by the Human Ethics Committee of Victoria University of Wellington.

The experiment focused on collecting data about usability, engagement, use of the various features, and user behaviour in this environment. We will first summarise relevant results that feed into our thoughts on the design of the system and of block programming systems here; for further details of these results, and other results from the experiment that we will not rely on here, see the original study [5, 47]. In Section 6 we present a novel analysis of people's revealed preferences for different modalities and different tasks, analysing the published data set from the same experiment.

5.1. Procedure

Each participant first completed a pre-questionnaire about themselves before being given a brief introduction to the system. The experimental system was instrumented and all interactions recorded. There were a total of six tasks in the body of the experiment, presented one at a time by the experimental system:

Task	Initial	Description
0	Tiled	Warmup – discarded
1	Tiled	Change Fibonacci to factorial
2	Tiled	Correct errors in this program
3	Tiled	Swap behaviours of two objects
4	Text	Describe program without running
5	Tiled	No specific goal – finish at will

The tasks were chosen to cause every participant to encounter both views and the error reporting at least once, and to have them both understand and modify code. Task 5 was intended to measure implicit engagement, giving no set task but telling participants that they could continue to use the system if they wished, and move on to the post-questionnaire when ready.

5.2. Summary

We will briefly summarise relevant results from the experiment [5, 47]:

- Participants showed high levels of engagement on multiple metrics, including implicit engagement with the system once tasks were complete and Likert-scale feedback on the post-questionnaire.
- The ability to switch views was widely used, with the median participant switching six or more times and 75% at least four.
- One quarter of participants spent more than half their time in text mode, and one half of them spent less than a third of their time in text mode.
- More-experienced participants viewed the system less favourably than less-experienced participants, as shown in Figure 7.
- The error reporting overlay was the aspect most often mentioned positively unprompted, by one third of participants.

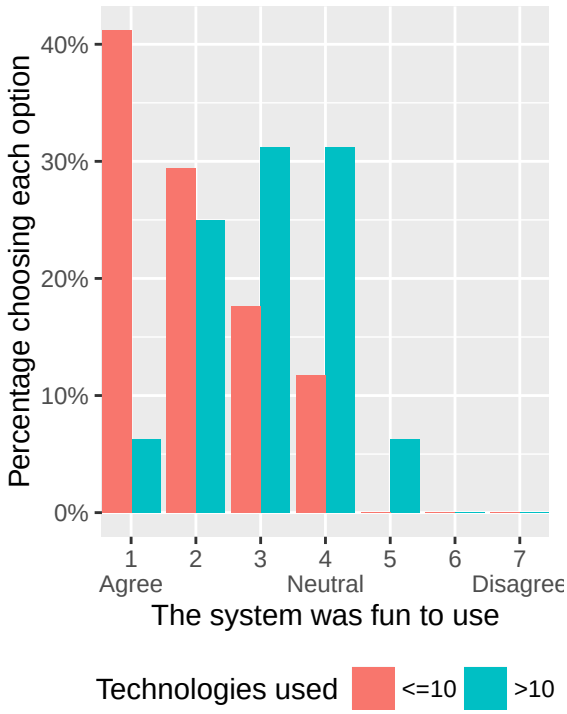


Figure 7. Participants' agreement with "The system was fun to use", one of a suite of engagement metrics in the experiment, split in half by a metric of experience relating to past programming and technological usage.

- Using both views in combination was also identified as helpful to understanding, mentioned by 9 of 33 participants and used by more.
- Many users found drag-and-drop frustrating, 40% mentioning it unprompted, and around 15% of participants had debilitating difficulty completing tasks with drag and drop.
- Most users did not switch to the tiled view in task 4, which only asked them to comprehend and describe the code.

This last point was not what had been expected while planning the tasks. Further analysis showed that of those 15 who did switch, half used tiles for more than 80% of the time spent on the task, and half for 35% or less, with nobody in between, and that while more-experienced users were more likely to switch, they were equally likely to be in either of those two groups.

This divergence of approach for a simple comprehension task suggested a difference in underlying preferences of modality between people, and led us to conduct the novel analysis presented in the next section, of how participants actually used each view during the editing tasks.

6. Preferred Modalities

The experiment with Tiled Grace provided a unique opportunity to examine which modality — blocks or text — users preferred for particular tasks. Spurred by discussions at the Blocks & Beyond workshop in 2015, we conducted an exploratory analysis of the collected data from the original experiment, which is publicly available [46], to see what trends might exist in the use of each mode.

We caution that, because of the overall sample size and the nature of breaking it down into further subgroups, these trends can only be suggestive of future research avenues. We did not have any particular hypothesis about what we might find, but these results will provide hypotheses for future research.

We focused on two areas specifically. First, we collected all changes made in "short" sessions in the tiled mode, which we defined as those logging ten non-automated events or fewer. We take these short sessions as representative of the user electing to perform specific actions in the tiled mode rather than as text. Secondly, we examined *all* text-editing sessions. In part, that is a technical limitation — unlike for the tiled mode, text operations were not discretely logged, only snapshots of the code — but we found it acceptable because text was not the default mode for any of the analysed tasks, so any use of the text mode indicated a deliberate choice by the user. We considered analysing all tiled sessions as well, but large sessions inevitably involve assembling whole programs and are not informative for this kind of analysis, as they reduce the measurements to counting the number of drags, new tiles, and so on, that are required to complete the tasks. As a result, however, some text sessions are much longer in wall-clock time than a short tiled session could reasonably be, and it is possible that more complex operations can be performed in them. A larger study might be able to tease out more detail from such data, and this exploratory analysis may suggest avenues to focus on. No participant spent all of their time on a task in the text view, while several spent all of their time in the tiled view. In both cases, we did not include tasks 0 (warmup) or 4 (comprehension only) in the analysis.

6.1. Editing as Text

We manually coded all 83 text-editing sessions by comparing the code before and after. Where the user made changes, and then undid them either themselves or through using our revert-changes option after a failed compilation, we recorded that fact mechanically.

Each session could be assigned multiple codes if more than one operation occurred during the session, but ancillary modifications as part of a broader code were not included (for example, "cut and paste" does not entail "delete"). A "before" and "after" snapshot of the code was mechanically obtained for each session and the two were compared by hand. The first author assigned each change a code or codes, in consultation with a colleague, after a first pass identifying candidate codes such that all present modifications were covered by a code. The initial set of candidate codes was taken from the codes

Table 2. Proportions and frequencies of codes of text-editing sessions. Each session could be assigned more than one code. Exp indicates the proportion of users in this code who were in the more-experienced half of the sample. 53% of all text sessions resulting in changes were by more-experienced users.

Freq.	Code	Prop.	Exp.
39	No change	47%	57%
4	Accepted offer to revert	5%	100%
4	Made and undid own changes	5%	67%
10	Change value of string/number	12%	57%
8	Cut and paste	10%	71%
8	Assemble complex code	10%	57%
8	Delete code	10%	43%
6	Change operator	7%	20%
6	Copy and paste	7%	83%
5	Subvert tiled error checking	6%	50%
3	Rename a variable	4%	33%

established for tiled sessions in the next section, and extended for text-only operations. Codes for operations that could only exist in tiles, such as drags, were also deleted.

47% of all text-editing sessions (39) made no change to the code, and seem to have been "just looking". We had earlier hypothesised that users may have found the ability to look at the code in two ways useful simply to break the monotony and to get an overview of the code, which we meant primarily as users switching from text to tiles to see the structure manifested graphically, but it appears that this may have been the case in both directions. This result may be a point in favour of "dual view" visual languages that display text and a visualisation simultaneously. 21 of 33 users (64%) had at least one empty text session.

43% of all text-editing sessions (36) had modified the code at the end. 15 of 33 users (45%) had at least one such session. The remainder of text sessions undid the changes they had made in one way or another. Table 2 shows the frequency and proportion in each code. Sessions may be assigned multiple codes.

The single most common modification was to *change the value of a string or a number*. Making such a change in the tiled view is among the easiest operations to perform as tiles, so switching to text to perform it seems a very deliberate choice. Most commonly, participants were making multiple such modifications at once, which may suggest a slight preference for textual interaction for repeated similar tasks.

The second most common operations were *cut and paste*, *deleting code*, and *assembling complex code*, in a three-way tie. By *assembling complex code* we mean producing a modification that would involve multiple drags and drops into holes. *Cut and paste* is exactly the operation that dragging and dropping tiles performs, so switching to text to perform it is not an obvious advantage. *Deletion* is also supported by the tiled interface, but deleting a single line from within a block of code was likely easier as text.

We believe that *changing operators* is an artefact of our implementation, rather than a meaningful result. The system sometimes interpreted attempts to select a different operator in an arithmetic tile as very short drag-and-drop sequences, which was a problem noted in the experimental results.

The most interesting task people performed in text mode was *subverting tiled error checking*. Although rare, it appears from subsequent analysis of the context that users frustrated with being unable to make logical, structural, or type errors in the visual interface switched to text in order to introduce the code they wanted (for example, adding two strings). This highlights an asymmetry in our system: the text and visual editors enforce slightly different rules on their code, which may have led to confusion. On switching back to tiles, these programs would have immediately reported errors.

6.2. Editing as Tiles

We categorised short and long tiled-editing sessions mechanically, discarding the long ones. Short sessions had ten or fewer logged interaction events. We also discarded short sessions at the beginning of Question 2, as this question presented a broken program that required a small degree of fixing before it was possible to switch views, and empty sessions at the end of tasks. Following this, 30% of all tiled sessions (70/234) were short.

37% of all short tiled-editing sessions (26) performed no operations at all, and seem to have been "just looking". These users may be viewing the structure of their textual program with a visualisation to help them to understand what they were doing. 12 of 33 users (36%) had one or more tiled sessions with no modifications.

The remaining 44 sessions performed some operation. In contrast to the text sessions, we also recorded interactions with the system that did not cause modifications to the program (no such interactions were available in text mode). Each session was coded for each operation that occurred within it, so some sessions were coded in multiple categories. Final coding was performed mechanically, after iterated manual inspection of uncoded segments created codes and rules to mechanise them. Table 3 shows the proportions of these sessions in each category. 19 of 33 users (58%) had one or more non-empty sessions.

As for text editing, the single most common operation to switch to tiled view to perform is *changing the value of a string or number tile*. It appears that this innocuous, but ubiquitous, operation is strongly polarising: it is a task that people will both switch to text to perform, and switch to tiles to perform.

The second most common operation is to *drag a single block into a hole*. This operation corresponds to textual cut-and-paste, moving code from one place to another.

Switching panes was counted if it occurred two or more times in combination with other operations other than creating a new tile, or was the only operation that occurred in the session. These sessions indicate users looking through the available functions to find the one they wanted, and are likely

Table 3. Frequencies and proportions of short tiled-editing sessions in each category. Exp indicates the proportion of users in this code who were in the more-experienced half of the sample. 37% of all non-empty short tiled sessions were by more-experienced users.

Freq.	Code	Prop.	Exp.
26	No change	37%	42%
13	Change value of string/number	19%	36%
9	Drag one block into a hole	13%	38%
7	Switch panes in toolbox	10%	67%
5	Meaningless drag	7%	50%
5	Rename a variable	7%	60%
3	See variables in scope	4%	33%
3	Assemble complex code	4%	0%
3	Fix code broken in text mode	4%	33%
1	Append one block to another	1%	0%
0	Delete code	0%	0%
0	Create new tile from toolbox	0%	0%

to be users primarily using text who wanted to know which methods were available. Some "no change" sessions may reflect the same use. The provision of a similar toolbox in the text mode, as in the work of Price & Barnes [48], would likely avoid these sessions.

Assembling complex code was naturally difficult within the restriction of a short session (measured in interaction events). The incidence of this code may not be informative.

Zero short sessions included dragging a tile out of the toolbox into the workspace.

6.3. Summary

Modifying the value of a string or number was the single-most common task performed in both modalities, in both cases commonly as the *only* operation performed in the session. This suggests to us that users have divergent views on the appropriate way to perform this action when given the choice, but we have no hypothesis as to why. We cross-referenced these codes with the original experiment's division of participants into more- and less-experienced halves; 62% (8/13) of such tiled modifications were by less-experienced users, compared with 40% (4/10) of the textual modifications. These proportions are roughly in line with the relative usage of the modes by each group found in the original experiment. There is an evident difference, but neither is overwhelming and given the sub-sample size nothing definitive can be said.

A plurality of sessions in both modes were empty, where the user performed no actions and simply looked at their code. These sessions were common among both more- and less-experienced users. We take this as partial validation of the hypothesis put forth previously [5, 47] that simply having two drastically different views available is found useful in itself.

Participants attempted to use the other view to work around limitations of the principal view they were using, for both good and ill. The interaction with operator tiles was a flawed design, and many users used the text view to make modifications they had difficulty with in the tiles, which was a positive use of the functionality. The enhanced error checking possible in the tiled view, which prevented many type and structural errors even being introduced, was an obstacle to some users who then used the text view to construct their broken programs, a negative use of time that could have been spent establishing *why* the code was not allowed. Neither technique was the intended use of the system, however, and both represent flaws or at least limitations in the approach taken by Tiled Grace.

Many participants opted to use textual cut and paste for operations that could easily be carried out by drag and drop. It may be that users are conditioned to perform such "moving" operations as text, which may also explain the high proportion of string or number modifications performed in the text mode.

Further research is required to confirm or refute all of these findings, particularly those with no a priori reason to expect them.

7. Discussion

A very common issue that has been encountered in introductory visual languages is that learners do not consider them "real" programming languages [30, 39, 40]. In some cases, the fact of being "easier" than text was interpreted to mean that the visual language did not "count" as programming, particularly when textual programming was later found challenging. Preconceived ideas about what is and is not programming, or what kinds of programming are or are not useful, can lead to block-based systems being regarded negatively by both current and prospective users. DiSalvo [49] found that learner perceptions of visual and textual programming systems varied according to the ultimate career goals of the user. Users with an interest in a programming career were more inclined towards a textual language than Alice, while those interested in media and design careers inclined the other way. For some, finding the textual language harder was in fact a point in its favour.

In Tiled Grace we aimed to avoid the perception that a block-based language was not "real" by having the block and textual language exactly match one another and presented equally. Experimentally, we did see users deploy both modes, and seem to understand the connection between the two. Similar to DiSalvo [49], and like Weintrop [50] and Matsuzawa [20] as well, however, we found that a number of participants very strongly preferred the textual mode and were even scornful of the tiled mode and its presence, to the point of using text almost exclusively even when the system design made it more difficult to complete tasks in that way. Three participants explicitly noted unprompted that they were predisposed to dislike GUIs. Other users exclusively used blocks. On one level, the fact that they were able to do so within the same system speaks to a more inclusive environment than a single-paradigm approach allows. On another, it seems that the presence of the other mode was at best neutral and possibly a net negative in overall perception for both groups of users, and might be off-putting. Neither group was more than 15% of users in our study. We did not ask about future goals in our experiment, but our more-experienced users (representing

the closest proxy we have to those whose self-conception is as programmers) were notably less positive about the system than less-experienced users. Our — somewhat informal — sense from observation and feedback in the experiment is that perceiving the block language as a toy, and thus the system as a whole as one too, played into this view. These perceptions may be a concern for the targeting and marketing of block languages.

Our experiences with Tiled Grace are paralleled by the experiment of Weintrop and Wilensky [27, 51] about multi-modal block-text environments. Their study involved high-school students in three conditions (blocks only, blocks and read-only JavaScript text, blocks and modifiable JavaScript text) who began in Snap! [22] and subsequently moved on to (textual) Java, over a ten-week course.

One facet of the study involved student performance over different concepts and modalities [52], having been exposed to both graphical Snap! and textual Java across the course and completing commutative assessments with parallel questions. For most concepts, students performed better with graphical code than textual. Students performed equally well on comprehension tasks regardless of modality, while in our experiment when given a pure comprehension task and a choice of views, most participants did not deviate from the default text view, despite overall preferring the tiled view in the experiment as a whole.

There may be no particular advantage in understanding code to either a block or textual view, despite blocks making the structure of the code explicit, at least for the relatively small programs in use in both experiments. What advantages do accrue from block editing would then all come from easier construction of programs, rather than easier understanding. Such a result does not seem to match with the self-reporting of participants in either study or elsewhere in the literature, and so needs further investigation.

When Weintrop's students were asked to compare Java and Snap!, those who expressed a view overwhelmingly said that block-based programming was easier than text, regardless of which condition they had originally been in. In our experiment, similarly, participants regarded the tiled view as somewhat easier to use, although not by the same 80% margin. Weintrop was able to conduct interviews with participants to attempt to establish the reasons behind these results, which can shed some light on our own findings.

Weintrop's analysis of the interviews in the study investigates (among other things), just *why* blocks were found easier. A number of points correspond to our study, and to how we found that participants used each mode in Section 6. One that did not, however, was that "blocks are easier to read" because the *language* of blocks was different and more "English". We believe that this limitation is quite significant: to find out whether blocks, themselves, are helpful, the languages should match as closely as possible. With Tiled Grace, where they match exactly, we did find that users regarded the tiled view as making it easier to deal with syntax, matching those from the Weintrop experiment who mentioned punctuation, balanced brackets, and other syntactic noise as reasons they found blocks easier. It is important to be careful not to conflate elements of a block language with elements of the block paradigm itself (albeit that this is very difficult to avoid with current systems).

Our blocks did not have different shapes for different data types, but were approximately colour-coded by topic (for example, variable declarations and variable references were similar shades). We had wished to make them shaped to complete the "jigsaw puzzle" metaphor, but were unable to do so with the wide range of types possible in a general-purpose language. The Weintrop study finds that these shapes were one of the key reasons that users said they found blocks easier to use: the shapes of a block and hole communicate whether they are compatible, while top and bottom connectors made sequencing explicit. Our type checker would disallow many invalid combinations, but only after the user had tried to perform it, and in some cases they would then switch the textual view (which had less stringent immediate checks) simply to create the code they wanted. It is possible that, had the blocks been obviously incompatible, users would not have attempted this to begin with. On the other hand, as Weintrop found, the puzzle-piece metaphor can lead to confusion among learners who expect there then to be "a" solution to a problem, as in a jigsaw, rather than many possibilities. This expectation did not noticeably appear in our experiment, but some participants did express that they thought they had completed some tasks "wrong".

Weintrop and Wilensky's other two reasons, that composing code was easier or more accessible as blocks, and that blocks were memory aids, were both borne out in our study. The pane-switching tiled sessions from Section 6, and likely some of the empty sessions, appear to be exactly using the block side as a memory aid. We also observed "bottom-up" construction of complex expressions to be common, often using an out-of-the-way corner of the workspace to build up the expression before moving it into place. Because Tiled Grace enforced scoping of variable-reference tiles (necessary, as the textual language has traditional lexical scoping, and in fact intended to help by offering a list of available names), assembling code in this way was sometimes not possible, to the frustration of the user. One of the trade-offs in integrating the textual and block languages that we had not considered was that this sort of "inside-out" construction, which is very natural and widely-reported [53] in block languages, would be stymied by error prevention in the textual language. Meerbaum-Salant *et al.* have argued that this style is in fact a "bad habit" [53], and that it has a longer-term detrimental effect on learners. It is not obvious whether this aspect of Tiled Grace is helping or hindering, and precisely what the long-term goals are may again be important.

A later experiment by Weintrop and Holbert [50] used Pencil Code as a switchable dual-mode environment, with the goal of finding out how each mode was used, as in our analysis in Section 6. Unlike in our results, the majority of switches to blocks were to add new blocks to the program, and empty block sessions were no more than 5.7% of the

total. Moving a block inside another was somewhat common in both experiments. While we considered short sessions as a whole, Weintrop and Holbert looked only at the first action taken after a change of modality. Given this, it is remarkable that so many more sessions created new tiles. Weintrop and Holbert note that two-thirds of the time a new block was added in this way, and 86.7% of the time a control block was added, it was the first time that block had been used. The nature of our experiment, where starter programs were already provided, may have limited the number of occasions to add a block in this way. The most significant difference between the subject populations for our experiment and theirs is that our users were adults who primarily had past programming experience, while Weintrop and Holbert's were a mix of novice learners in high school and graduate students outside of programming (the groups are not separated in this part of the analysis). It may also be that these populations are the cause of these different observations, or that the nature of the language affects user behaviour in some way (in particular, Tiled Grace had a significantly lower total number of distinct blocks available).

Similar to in our original experiment, Weintrop and Holbert find a wide range of levels of use of each modality, with a trend for increased text use to go along with higher degrees of experience. They also note some users who strongly prefer either blocks or text almost exclusively, as did we. These results are in line with the goals of Tiled Grace's design, and Weintrop and Holbert suggest that they provide support for the dual-modality approach as providing for "low-threshold/high-ceiling" programming environments.

Matsuzawa *et al.*'s experiment using both Java and Block, which translates to and from Java, in an introductory programming course finds a wide range of different levels of use of textual and block editing [20]. The BlockEditor system supports exporting to Java and importing from Java, but makes no particular explicit mapping between the two in itself; it does not appear that this caused any widespread trouble for learners, which may suggest that Tiled Grace's emphasis on making the mapping manifest through animation is unnecessary, or it may be a reflection of the teaching structure employed.

Across a fifteen-week course students tended to use less of the block view as time progressed, but with highly varied rates of change between students as well as individual fluctuations. The rate of backsliding and inter-student variation supports our goal of making the transition be a process, rather than an event, as many students would have been left behind given any particular transition point. It is notable, however, that after a large task in the eigth week the rate of block usage did drop dramatically, and stayed low, so it is possible that there is a distinguished point where text becomes preferable. It is also possible that using the text view out of necessity, when the block view of a large program has become unwieldy, acclimatises learners to it, and simple exposure is all that is required to cause the text modality to take over.

Experiments using TouchDevelop with secondary-school students [54] found that students were rapidly able to develop non-trivial mobile applications in that environment. Long-term users were able to produce advanced applications with no formal instruction other than sample code, while shorter sessions showed good performance regardless of programming background. One posited explanation is that, because TouchDevelop's tap-based interface surfaces the available actions in a given context on demand, it promotes experimentation with a wider range of options with immediate feedback. Similarly, users will less often need to search for the block they want to use if TouchDevelop presents what it expects they may need automatically. The "memory aid" activities from our and other systems should not need to occur in TouchDevelop, and some sort of context-aware suggestion mechanism would be an advantage to the user of a block language.

Our experiment found that a sizable proportion of participants (around one in six) had debilitating trouble using drag-and-drop interaction with the system, despite everyday use of mice and keyboards. Tiled Grace relies on the mouse pointer being over a drop target, which we had taken as the standard drag-and-drop behaviour, and these users found that task very difficult. Other contemporary block languages have similar behaviours: Scratch and Blockly use a point in the upper left of the bounding box of a tile instead of the mouse pointer, while Pencil Code uses a similar point in combination with a Euclidean distance metric to choose the closest target. Neither of these seems obviously more intuitive. We have not seen studies reporting on this significant of a difficulty dragging in block-based languages, but have anecdotally observed tiles going otherwise than where they were wanted in all of these systems. Past human-computer interaction research [55–57] has found that point-and-click interfaces may involve fewer errors and be faster than drag-and-drop, although recent research with children [58] has shown that they may both expect and prefer drag-and-drop interfaces. Ludi [59] has noted accessibility problems with contemporary block languages for users with motor or visual impairments. These are a significant issue that is fundamental to the interaction paradigm most of these languages currently use, and which is only exacerbated by the observations in our experiment. Other block-like structured editing paradigms, as in Kölling *et al*'s "frame-based" Stride [25] language, or the "point-and-tap" TouchDevelop [24] interface that requires no continuous action, may be more suitable. Of the drag-and-drop approaches, Pencil Code's appears the most usable, but still relies on free movement of the mouse.

When Powers, Ecott, and Hirshfield experimented with transitioning from Alice to Java (with BlueJ) in an introductory programming course [30] 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. In our experiment, which used exactly parallel languages in both textual and visual modes, and in Matsuzawa *et al*'s [20], which did not, understanding the relation between the two did not seem to be an especial problem, but intimidation

by text was evident. Students in Matsuzawa *et al*'s study with lower self-rating of their skill avoided the text view, but none of our self-rating questions showed a strong correlation with use of either mode; we did not have a generic "rate your programming skill" question, however. Weintrop's earlier experiment appears as though a similar trend may exist, but explicit data is not available. As our experiment was much shorter than any of the others, the trend may not have had time to emerge.

8. Lessons and Conclusion

We see mixed success and weakness in our experiences and results working with block-based languages. We will attempt to distill some key lessons from our experience with Tiled Grace that are applicable to block languages more broadly. These lessons draw from our and others' experimental results and observations, and from our experience designing and building an integrated block-text programming system.

A positive sign is that our experiment, and others', showed strong engagement with a predominantly block-based environment, and that **even when given the choice to use text users in an unfamiliar language largely preferred to use the blocks.** This held even though most of our users had some familiarity with text programming in other languages already.

One reason that some block systems have been found easier to use is that the *language* of the blocks is more accessible, or more "English", than conventional languages. This is a language-design element, rather than a property of block systems, and is a natural confound when assessing how helpful a system is, so **it is important to separate the effects of the language and the interaction paradigm when evaluating block systems**. Tiled Grace's use of identical block and text languages is one method of keeping this distinction clear, but designing textual languages that incorporate the benefits of block languages is another approach.

We also found an approach to reporting errors in block-based programs that was effective and well-regarded by experiment participants, including unprompted positive mentions and strong signs of effectiveness at communicating the issue identified. This approach could easily be applied to other languages in the same model, while conventional textual languages would find it difficult to provide the same level of immediacy. **Immediate in-situ feedback is effective and much easier in block languages than text.** We recommend incorporating some similar form of error reporting into all block-based languages, unless there is a strong pedagogical reason to turn ill-conceived programs into runtime debugging exercises.

Less positively, we found that **more-experienced users were substantially less favourable towards a block-based environment than less-experienced users**. While for purely novice systems this may not be an issue, it is a significant caution for systems aiming at broader markets or professional use. Even novices will become more experienced over time, and losing engagement is a problem for, at least, retention.

We believe that supporting people to move on to other paradigms (whether through our and Pencil Code's dual-mode approach or otherwise) is crucial to successfully deploying block languages for programming education (while general-purpose or domain-specific block languages may not wish to do so). The experience-engagement results of the previous paragraph are one reason why, but more important are the educational psychology aspects discussed in Section 4.1. Thus, **block languages *for programming education* must have an exit strategy.** Course structures predicated on simply starting in a block language and moving on to a more conventional language after a few months are fraught with danger unless significant care — and time — is put into providing explicit bridging instruction to help learners map concepts from one world to the other. Languages that do not facilitate this process are doing their users a disservice, but allowing and encouraging mixed use appears effective.

While it is well-known that experienced users can find the back-and-forth dragging of Scratch and other block languages frustrating, it has been less noted that **drag-and-drop visual editing is a significant problem for some users**, even those without physical limitations on doing so. In addition, drag and drop is much less accessible than text editing for anybody unable to use a mouse easily, or to see what is on screen. If block languages aim to democratise programming, they cannot do so by excluding already-marginalised people further. Block paradigms that are not dependent on drag and drop may be more suitable for everybody.

Finally, we found that users made heavy use of our view-switching ability simply to see "the other side": they did not always want to make changes there. These results emerged from the instrumentation in our experiment and from free-text feedback. It is not all-or-nothing: **providing multiple views of code helps users be more comfortable with it**, even if the code is not edited (or editable) in one view or another.

Block programming is currently undergoing substantial growth, but we should not lose sight of potential negative aspects. Long-term thinking is required in their design and use, and experimentation to determine which aspects of them are helpful, and which are ancillary or negative.

References

[1] 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 *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 129–134.

[2] A. P. Black, K. B. Bruce, M. Homer, and J. Noble, "Grace: The absence of (inessential) difficulty," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! '12. New York, NY, USA: ACM, 2012, pp. 85–98.

[3] J. Noble, M. Homer, K. B. Bruce, and A. P. Black, "Designing grace: Can an introductory programming language support the teaching of software engineering?" in *Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference on*. IEEE, 2013, pp. 219–228.

[4] M. Homer and J. Noble, "A tile-based editor for a textual programming language," in *Proceedings of IEEE Working Conference on Software Visualization*, ser. VISSOFT'13, Sept 2013, pp. 1–4.

[5] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Proceedings of IEEE Working Conference on Software Visualization*, ser. VISSOFT'14, Sept 2014.

[6] M. Homer, T. Jones, J. Noble, K. B. Bruce, and A. P. Black, "Graceful dialects," in *ECOOP 2014 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed. Springer Berlin Heidelberg, 2014, vol. 8586, pp. 131–156. [Online]. http://dx.doi.org/10.1007/978-3-662-44202-9_6

[7] M. Homer, K. B. Bruce, J. Noble, and A. P. Black, "Modules as gradually-typed objects," in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, ser. DYLA '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:8.

[8] 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.

[9] A. Kay, "Squeak Etoys authoring & media," Viewpoints Research Institute, Research Note, 2005.

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

[11] Blockly Project, "Blockly web site," https://code.google.com/p/blockly/.

[12] D. Wolber, "App Inventor and real-world motivation," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 601–606.

[13] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in app inventor 2," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, July 2014, pp. 129–132.

[14] K. Chadha, "Improving the usability of App Inventor through conversion between blocks and text," Honors Thesis, Wellesley College, 2014.

[15] K. Chadha and F. Turbak, "Improving App Inventor usability via conversion between blocks and text ," *Journal of Visual Languages & Computing*, vol. 25, no. 6, p. 1042–1043, 2014, Distributed Multimedia Systems (DMS2014) Part I.

[16] P. Guo, "Proposal to render Android App Inventor visual code blocks as pseudo-Python code," https://people.csail.mit.edu/pgbovine/android_to_python/.

[17] D. Bau and D. A. Bau, "A preview of Pencil Code: A tool for developing mastery of programming," in *Proceedings of the 2nd Workshop on Programming for Mobile Touch*, ser. PROMOTO '14. New York, NY, USA: ACM, 2014, pp. 21–24.

[18] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, Jun. 2015.

[19] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 445–448.

[20] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-CS introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 185–190.

[21] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015.

[22] B. Harvey and J. Mönig, "Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists?" in *Constructionism 2010*.

[23] D. Blank, J. S. Kay, J. B. Marshall, K. O'Hara, and M. Russo, "Calico: A multi-programming-language, multi-context framework designed for computer science education," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '12. New York, NY, USA: ACM, 2012, pp. 63–68.

[24] 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.

[25] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '15. New York, NY, USA: ACM, 2015, pp. 29–38.

[26] N. C. C. Brown, A. Altadmri, and M. Kölling, "Frame-based editing: Combining the best of blocks and text programming," in *Fourth International Conference on Learning and Teaching in Computing and Engineering*, ser. LaTiCE '16, 2016.

[27] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: Students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 199–208.

[28] Playful Invention Company, "Picocricket reference guide, version 1.2a," http://www.picocricket.com/pdfs/Reference_Guide_V1_2a.pdf.

[29] S. Dasgupta, S. M. Clements, A. Y. idlbi, C. Willis-Ford, and M. Resnick, "Extending Scratch: New pathways into programming," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015.

[30] K. Powers, S. Ecott, and L. M. Hirshfield, "Through the looking glass: Teaching CS0 with Alice," *SIGCSE Bulletin*, vol. 39, no. 1, pp. 213–217, Mar. 2007.

[31] 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.

[32] D. N. Perkins and G. Salomon, "Teaching for transfer," *Educational Leadership*, vol. 22, no. 32, 1988.

[33] A. Robins, "Transfer in cognition," *Connection Science*, vol. 8, no. 2, pp. 185–204, 1996.

[34] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 213–229.

[35] J. L. Dyck and R. E. Mayer, "Teaching for transfer of computer program comprehension skill," *Journal of Educational Psychology*, vol. 81, no. 1, 1989.

[36] V. R. Delclos, J. Littlefield, and J. D. Bransford, "Teaching thinking through Logo: The importance of method," *Roeper Review*, vol. 7, no. 3, 1985.

[37] D. H. Clements and D. F. Gullo, "Effects of computer programming on young children's cognition," *Journal of Educational Psychology*, vol. 76, no. 6, 1984.

[38] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, p. 131–174, 1996.

[39] C. Lewis, S. Esper, V. Bhattacharyya, N. Fa-Kaji, N. Dominguez, and A. Schlesinger, "Children's perceptions of what counts as a programming language," *Journal of Computing Sciences in Colleges*, vol. 29, no. 4, pp. 123–133, Apr. 2014.

[40] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. Scratch," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 346–350.

[41] T. Jones, M. Homer, J. Noble, and K. Bruce, "Object inheritance without classes," in *30th European Conference on Object-Oriented Programming (ECOOP)*, 2016.

[42] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 132–141.

[43] T. Jones, M. Homer, and J. Noble, "Brand objects for nominal typing," in *29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[44] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic blocks: Formalism-inspired ui for structured connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 3063–3072.

[45] M. Vasek, "Representing expressive types in blocks programming languages," Honors Thesis, Wellesley College, 2012.

[46] M. Homer, "Tiled Grace experiment data," Available as `tiled-grace-experiment.tar.bz2` attached to [47] in the research archive of Victoria University of Wellington at http://researcharchive.vuw.ac.nz/handle/10063/3654., 2014.

[47] M. Homer, "Graceful language extensions and interfaces," Ph.D. dissertation, Victoria University of Wellington, 2014.

[48] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 91–99.

[49] B. DiSalvo, "Graphical qualities of educational technology: Using drag-and-drop and text-based programs for introductory computer science," *IEEE Computer Graphics and Applications*, vol. 34, no. 6, pp. 12–15, Nov 2014.

[50] D. Weintrop and N. Holbert, "From blocks to text and back: Programming patterns in a dual-modality environment," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: ACM, 2017, pp. 633–638.

[51] D. Weintrop, "Modality matters: Understanding the effects of programming language representation in high school computer science classrooms," Ph.D. dissertation, Northwestern University, 2016.

[52] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 101–110.

[53] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in scratch," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 168–172.

[54] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie, "The future of teaching programming is on mobile devices," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 156–161.

[55] K. M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children," *ACM Transactions on Computer-Human Interaction*, vol. 8, no. 1, pp. 1–33, Mar. 2001.

[56] D. J. Gillan, K. Holden, S. Adam, M. Rudisill, and L. Magee, "How does Fitts' law fit pointing and dragging?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '90. New York, NY, USA: ACM, 1990, pp. 227–234.

[57] I. S. MacKenzie, A. Sellen, and W. A. S. Buxton, "A comparison of input devices in element pointing and dragging tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '91. New York, NY, USA: ACM, 1991, pp. 161–166.

[58] W. Barendregt and M. M. Bekker, "Children may expect drag-and-drop instead of point-and-click," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '11. New York, NY, USA: ACM, 2011, pp. 1297–1302.

[59] S. Ludi, "Position paper: Towards making block-based programming accessible for blind users," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 67–69.