Blocks, Blocks, and More Blocks-Based Programming

Ben Selwyn-Smith
Oracle Labs
Brisbane, Australia
benselwynsmith@gmail.com

Craig Anslow
Victoria University of Wellington
Wellington, New Zealand
craig@ecs.vuw.ac.nz

Michael Homer
Victoria University of Wellington
Wellington, New Zealand
mwh@ecs.vuw.ac.nz

Abstract

Blocks-based programming is a common way to teach novices how to program. However, there are many block-based languages to choose from. This paper reviews Block-Based Programming Languages (BBPLs), takes a detailed look at a number of existing BBPLs including their features and comparing and contrasting these languages. Finally, through a number of research questions, this paper evaluates the current state of the art and points out areas for potential further research.

CCS Concepts: • **Software and its engineering** → *General programming languages*; • **General and reference** → *Surveys and overviews.*

Keywords: Block-Based Programming Languages, BBPL, Programming, Literature Review

ACM Reference Format:
Ben Selwyn-Smith, Craig Anslow, and Michael Homer. 2022. Blocks, Blocks, and More Blocks-Based Programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3563836.3568726

1 Introduction

In recent years programming has become increasingly popular as the number of people enrolling in Computer Science degrees has reached unprecedented levels [16]. One of the reasons for such an increase is the increasing technological ubiquity in everyday life, most importantly in the form of smartphones, with a lot of children receiving or at the very least being frequently exposed to a phone or tablet from a young age. Another reason is the increasing demand for graduates with computer science degrees, with a higher employment rate and above average salary potential [9], making pursuing a computer science degree an attractive choice for many prospective students. Similarly, there are concerns that jobs will be lost to automation as the potential of Artificial Intelligence (AI) continues to increase, however research has shown that computer science and engineering fields are one of the least likely to be replaced within the next 15 years [4].

Therefore, it is important to use the best methods for teaching computer science to new learners and research has shown that an effective way is to use a Block Based Programming Language (BBPL) [P16]. BBPLs are a type of visual programming language that use block shaped objects to represent programming concepts and operations. Users are then able to join these blocks together, to create functioning programs. Compared to traditional text based languages, BBPLs have many features that make them more suited to new users. The key amongst this paradigm is the removal of syntax errors, as blocks are always correct representations of existing syntax, and BBPLs prevent users from combining blocks in incorrect ways, either directly or through error messages, before a program is run. Another key feature, is the exploration potential of BBPLs. In textual languages when a novice wants to attempt something new, it is unlikely they will be able to guess how to construct the code to accomplish this, and it is likely that they will have to consult an external resource of some form. However, in BBPLs a user can instead place blocks on the screen and try combining them in various ways until something works, and while one could argue that this is less efficient in the short term, it is of great use for novice learners that might otherwise be too intimidated by a programming language textbook and therefore leads to greater gains in the long term. This is especially relevant as an increasing number of younger users are being introduced to programming through the adoption of mandatory programming courses for children in school, starting from as early as the elementary level [19]. BBPLs benefit younger users by the ease at which they can create interactive multimedia animations and games which are relevant to their interests [15].

Like traditional text based languages, they are many BBPLs and new ones are frequently being created for much the same reasons. Whether it be to try out a new potentially beneficial feature, remove potentially harmful ones, or a new design to cater to a different audience, there is seemingly always room for more BBPLs. Therefore, to help future BBPLs, this literature review aims to gather information on a large number of existing BBPLs, report their basic features

PAINT '22, December 05, 2022, Auckland, New Zealand

and comparatively examine them. The hope is that this review can act as a BBPL resource, confirm existing benefits of BBPLs, and highlight areas where more can be done.

2 Method

To the best of the authors' knowledge there exist no other literature reviews that cover the BBPLs for the same purpose as this paper and this was partly the motivation behind creating this literature review in the first place. However, one existing literature review that does stand out is by Moreno-León and Robles [12], which examines a single BBPL, Scratch, and related literature that investigate the potential for Scratch to be used to teach skills other than coding, such as Mathematics, English and so forth. However, we are interested to understand how BBPLs have been used for teaching programming.

While few restrictions were placed upon BBPLs being examined by this literature review, the goal was still to try and list only those that are currently relevant in some way. This means BBPLs that were either created recently or ones that still see regular use or have recent papers relating to them despite being older are included in this review. A detailed breakdown of the BBPLs chosen is included in section 3.1.

The process through which this literature review was conducted derives from the study of several resources on the creation of systematic literature reviews, most prominently: "Guidelines for performing Systematic Literature Reviews in Software Engineering" [7]. However, it must be noted that this particular paper is just a literature review and not a systematic literature review. The largest difference between the two in this paper is in the selection process for the related literature. Unlike a systematic literature review where the aim is to find all relevant papers and perform exclusion and inclusion analysis to find the most relevant, the method used in this paper is to simply find and read as many relevant papers as possible within the allotted time frame.

To find papers, various search strings including but not limited to: block(s), based, block(s)-based, programming, coding, code, language and languages, were entered into Google Scholar and online databases such as the ACM and IEEE Xplore digital libraries. However, despite trying many different combinations of these words, including larger search strings using AND and OR functionality, the number of relevant results retrieved was always very low. Instead, the method of choice for finding relevant papers was to start with a well known paper and perform forwards and backwards snowballing from there. The choice of well known paper to start from was an easy one: "Scratch: programming for all" [P28] has over 1500 citations according to Google Scholar. Papers that were part of the Blocks and Beyond workshop [18] from the VL/HCC 2015 symposium were also examined.

Like systematic literature reviews this paper started with some research questions to answer as part of the process. However, rather than reject papers that strayed too far from these questions, the questions were instead expanded to encompass found papers providing the papers were still within the general theme. This ultimately led to the following research questions:

- **RQ1** What block based languages exist?
- **RQ2** What textual programming languages do block based languages support?
- **RQ3** How have block based languages been evaluated?
- **RQ4** What educational settings have block based languages been used in?
- **RQ5** What computer medium have block based languages been deployed on?
- **RQ6** What kind of debugging features do block based languages have?
- **RQ7** What kind of influence have block based languages had on novice programmers?

3 Findings

This section firstly examines the findings from the papers reviewed as part of this literature review in terms of encountered block-based programming languages and their features. Visual examples of a selection of these can be seen in **Figure 1**. Secondly, it provides answers to the research questions previously listed. The full list of reviewed papers can be found in **Appendix A**.

3.1 Block-based Programming Languages

3.1.1 Scratch. Scratch is possibly the most well known of BBPLs, with close to 20 million registered users and having recently entered the top 20 most used programming languages according to the TIOBE index [17]. Scratch is a BBPL that allows novices to create programs that primarily manipulate two-dimensional images, referred to as sprites. Existing blocks either contain common programming functions such as loops, if statements and so on, or functions that can be performed on sprites, such as move 10 pixels down, or rotate 45 degrees clockwise. Scratch also supports user created media content, including a built in sprite editor and the ability to import audio files. This combination of features lead to Scratch being used for creating games and other entertainment focused media programs. Scratch was designed with the users in mind, with a strong focus on exploratory programming, which makes it very accessible even for younger users. Furthermore, to make things easier for novices, invalid blocks are skipped when executing, allowing invalid programs to still function. It also allows users to edit blocks while running, further improving the novice's understanding of their created program. Scratch does not support editing code through text and all programs exist entirely of blocks.

For debugging, Scratch features the ability to run specific groups of blocks, triggered by clicking on them. Scratch also

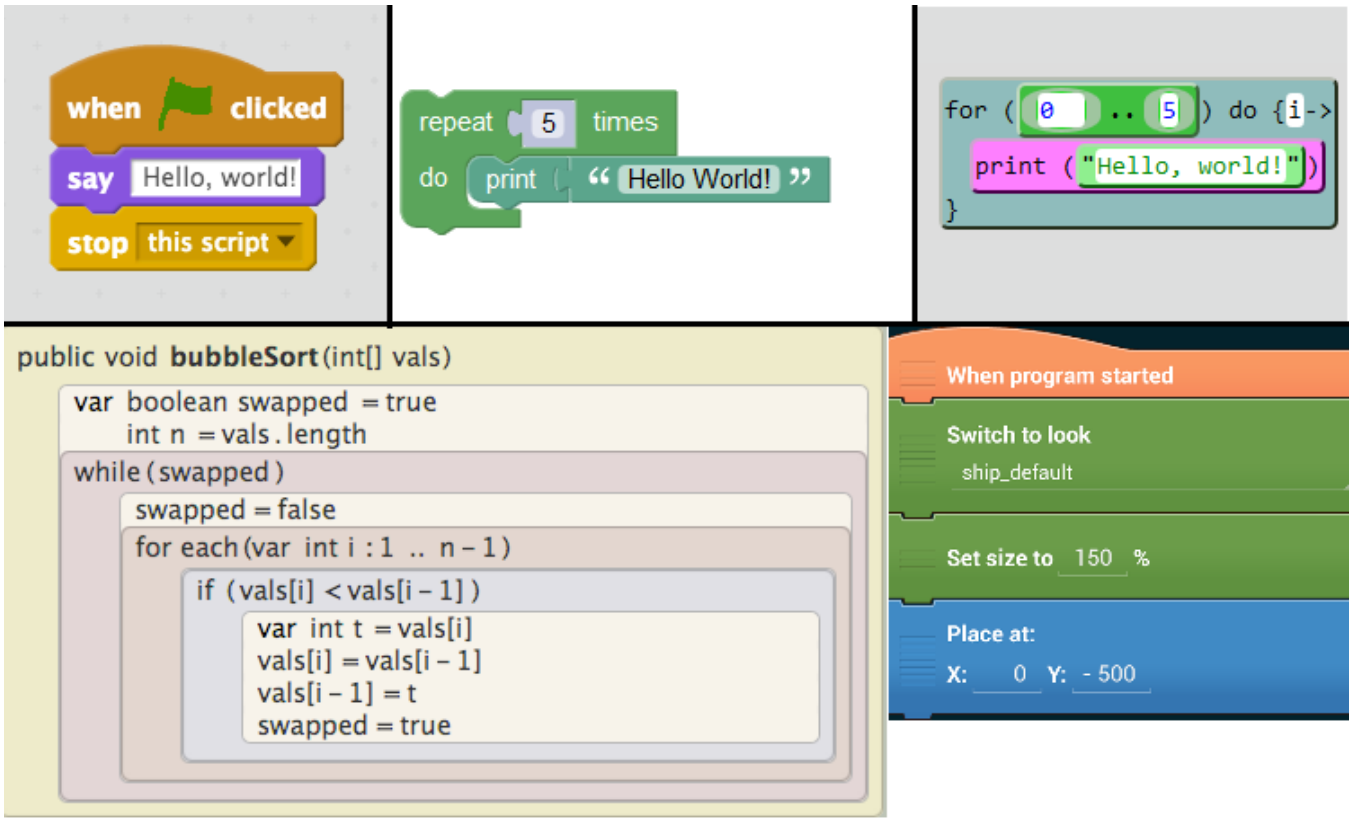

Figure 1. Some examples of BBPLs. Top Row: Left - Scratch, Middle - Blockly, Right - Tiled Grace. Bottom Row: Left - Stride, Right - Pocket Code

allows the value of variables to be checked during execution by use of a *say* block, which acts like a typical textual language *print* statement, that displays the chosen variable on screen. To help prevent errors and by extension the need for debugging, Scratch uses different shaped corners for blocks and holes, which denotes where certain blocks can be placed, these are sorted by data type, with rectangles for strings, rounded rectangles for numbers, and hexagonal blocks for boolean values. Trying to place a rounded block into a hexagonal hole for example, is not possible. Finally, Scratch has the ability to create entirely new blocks, which may contain any number of labels and number, string or boolean holes. This feature allows for some flexibility, but non data blocks cannot be added to new blocks. For example, you cannot create a new block that contains a while loop block.

3.1.2 Blockly. Blockly [P28] is very similar to Scratch, with coloured jigsaw like blocks that can be combined to make programs. Scratch focuses on media based story-telling, while Blockly mainly exists as a block library, allowing developers to make use of its features in their own software. Blockly also does not allow textual editing, but it does let users export their created programs into a selection of text languages, however this is strictly one-way, with no direct visual guide highlighting which block produces which section of code, leaving the discovery of the relationship between the two representations up to the user's own experimentation. When choosing to include Blockly as part of their software, developers can also choose to restrict the possible output languages and even choose to support a new one. Similarly, new blocks can be added as needed.

Blockly does not use different shaped blocks like Scratch does, but instead checks for invalid combinations when blocks are combined. For example, when trying to create a block that compares a number to a string, when placing the second value block, the first will be ejected. In this way novices are made aware that what they are trying to accomplish is not possible. However, there are limitations to this method, as when a novice user wraps one of these values in a variable, no error is raised. Like Scratch, Blockly does permit users to selectively execute blocks, but unwanted blocks must be disabled first, as all active on screen blocks are executed at run time. Lastly, Blockly features the ability to collapse blocks, where multi line blocks collapse into a single line, and single line blocks are reduced to mostly text. This feature helps save on screen space, which is always a

concern with BBPLs, as blocks often require a lot more space than their textual counterparts.

3.1.3 App Inventor. App Inventor [P35] is a BBPL designed to allow novices to create programs for the Android operating system. App Inventor is very similar to Scratch and uses the Blockly library as it's base. Compared with the base Blockly library, App Inventor includes considerably more blocks. A lot of these extra blocks are designed to interact with the various elements of created Android Apps, such as user interface (UI) element manipulation and phone sensor interaction. In addition, App Inventor expands the various categories from Blockly, with more features, such as additional list command blocks and so forth. App Inventor does not support text based editing, but some work has been done to try and convert the source code from a program to Java and Python, however neither has been officially completed. Additionally, TAIL [P7] is an extension to App Inventor that allows for the conversion of blocks to a block-text hybrid, which produces blocks with code inside them. This still does not allow for full conversion from block to text, but is designed as a potential intermediary step.

As App Inventory uses the Blockly library, the debugging features are largely the same. One difference is that App Inventor does allow individual blocks to be executed at will. However, as App Inventor programs are designed for Android phones, programs cannot be tested without running an Android emulator or having a connected Android device.

3.1.4 Alice. Alice [3] differs from other BBPLs in that it allows users to create programs in a three-dimensional space. However, it can be likened to Scratch, as blocks are used to manipulate three-dimensional models within this space, just as Scratch is used to manipulate two-dimensional sprites. Alice also supports object-based programming and event driven programming, but does not support textual editing, except via one-way export in to Java code.

Alice is designed to prevent errors by forcing users to choose block combinations from a list of acceptable blocks. This prevents invalid combinations, reducing the need for debugging. However, Alice does not feature any other debugging methods and when an error does occur, displays a generic error message suggesting the user submit an error report.

3.1.5 Pencil Code. Pencil Code [P5] is similar to Scratch in that it allows novices to make games and stories through blocks. Unlike Scratch, Pencil Code allows users to work using blocks or text and switch between the two at will. This transition features an animation helping to show the user how the two different modes match up. This is done through the Droplet [P4] extension to Pencil Code that was later built in. The text code used by Pencil Code can be either CoffeeScript or JavaScript and as such blocks represent features of these languages. Pencil Code also allows novices to edit the block workspace through the use of a block based HTML editor.

In Pencil Code, invalid blocks are not ignored and will cause errors. Errors are produced at runtime with execution halting at the first invalid blocks, then instead of seeing the expected program output, an error message is displayed. Pencil Code does not allow subsets of current blocks to be executed, as every on screen block must be part of the structured program, but blocks can be dragged outside of this area to effectively disable them. While using Pencil Code users can choose to switch between JavaScript and CoffeeScript at will, but blocks are mapped directly to the textual language in use and are not updated when switching, potentially putting valid programs into an invalid state.

3.1.6 Tiled Grace. Tiled Grace [P13] is another BBPL that allows novices to work with either blocks or text. Unlike Pencil Code, Tiled Grace is built upon the Grace language, a language developed specifically for novice learners [2]. Tiled Grace allows the user to switch between text or block view as long as the program is valid, with an animation directly showing the mapping between each block and textual equivalent. Compared to other BBPLs Tiled Grace features more advanced blocks, including Objects, Classes and Inheritance. Using *dialects*, any number of additional blocks that represent Grace code can be added.

The debugging features of Tiled Grace are also more extensive compared with other BBPLs. For example, when trying to combine invalid block combinations in Tiled Grace, an error message is displayed when the action is prevented to help users understand what went wrong. Also, as the user changes blocks or text, Tiled Grace checks whether the resulting program is valid and the UI includes a notification showing whether it is or not. Then when the user hovers over this UI notification, they can see detailed error messages of each error currently in the program. Lastly, Tiled Grace features the ability to highlight all instances of a variable or method name in the program, as found in some programming IDEs.

3.1.7 BrickLayer. BrickLayer [P8] is a BBPL designed to allow novices to learn syntax through blocks, rather than bypass the need for it like Scratch. To do this BrickLayer uses a side by side view of text and block displays, with matching sections from each being highlighted in the same colours, allowing users to clearly understand the mapping between both. Users can edit either section and see the relevant changes that take effect in the other. The text language used by BrickLayer is C. Compared with other BBPLs, BrickLayer helps novices learn the intricacies of control blocks such as *if* statements, by including in the block list, blocks for terminating and continuing these sections. Therefore to form an *if* statement properly, requires the user to realise the need for each individual component. Unfortunately, the extent of BrickLayer functionality, especially in terms of

debugging could not be examined, as a publicly available version of the software does not seem to exist. It should also be noted that this BBPL is not the same as Bricklayer, despite the very similar name.

3.1.8 Madeup. Madeup [P14] is a textual programming language that includes a block based interface, which has been designed to allow users to construct 3D objects through programming and uses the Blockly library. These 3D objects can also be printed using a 3D printer. Shapes are created in a Logo style turtle graphics manner. Madeup supports direct editing of text and blocks, however, Madeup is currently still in development and the version examined in this review had some issues with information disappearing between the block and textual views.

Madeup uses an extension of Blockly for its block view and therefore possesses the same debugging features. In addition to these features it also has a real time error reporting output window that is updated whenever the blocks or textual code change. Madeup also introduces the ability to change block statements into expressions, and this feature allows for much greater flexibility when designing programs. However, in the current version this allows for some unusual block combinations, such as a *print* statement within another *print* statement. It remains to be seen how this particular feature will function in the release version of Madeup.

3.1.9 GP. GP [P22] is a BBPL that has been designed for the purpose of eliminating the need for users of BBPLs to transition into textual languages, by allowing all programming tasks to be done using blocks. Part of this paradigm is that GP itself is largely written in the GP language. GP as a BBPL allows users to create interactive media based programs much like Scratch and other languages, and its blocks are in fact extended from Scratch. GP has a unique feature which allows the user to switch between block and textual view using a slider, which ranges from a pure block view to a pure textual view and covers all in-between states. However, even in the pure text view, text elements are still functionally blocks. Blocks can be edited using an experimental text edit mode, however the authors of this paper experienced limited success when trying to use this feature, with text edits often producing no change in the related blocks.

By default, GP uses block ignore when executing, so invalid blocks do not prevent a program from functioning. GP also features a developer mode, which when activated allows users to view detailed error messages, move through the function stack, and perform step by step evaluation of the code. Like Scratch, it also features shaped data blocks, preventing invalid combinations.

3.1.10 BlockPy. BlockPy [P3] is another BBPL that uses the Blockly library. It features both text and block view, with its text language being Python. One of the main bonuses of BlockPy is the use of Python libraries, which allow for easy manipulation of data and displaying of graphs, with one of the stated goals being "Data Science as a First-Class Feature." The web client includes the ability to import various data sets from a predefined list, including Amazon E-book sales, data on some of the world's most wealthiest people, worldwide earthquake data, to name just a few. However, the ability to import user generated data sets does not seem to exist.

The debugging features of BlockPy are more advanced than those found in the Blockly library, with controlled bi-directional line by line execution, which is also present in block mode although this is less useful as it corresponds directly to textual view lines. Additionally, BlockPy displays detailed error messages on execution, instead of ignoring invalid blocks. These debugging options are also very strict, to the extent that even low severity errors such as unused variables, which often result in just a warning, preventing execution entirely. Like Pencil Code execution is also stopped upon encountering any error, limiting the error feedback per execution to one error at most.

3.1.11 Calico Jigsaw. The Calico programming environment [P6] is designed around interoperability between different textual programming languages. The Jigsaw extension adds BBPL functionality to this environment. Block programs can be exported to Python and other languages in the Calico environment, but can not be imported, but Jigsaw does allow for parts of blocks to be edited through textual input. Calico shares libraries between all languages it contains, and with Jigsaw this means that a selection of block modules can be imported as required, providing a wide array of different blocks.

Jigsaw debugging features variable speed execution, including line by line execution, with breakpoints also supported. When executing, the value of blocks can be viewed and the program will pause and display an error message upon finding an invalid block.

3.1.12 BlockEditor. BlockEditor [P18] uses a block language called Block and can convert between blocks and Java. Editing can be performed in either mode. As BlockEditor uses a specific block language rather than a block representation of Java it is not a direct one to one mapping. However, the similarities are such that switching between the two poses no major issue. BlockEditor does not prevent users from changing views from textual to blocks when the program contains errors, as used by Tiled Grace, but instead prevents any blocks from being shown. BlockEditor also makes use of different shaped blocks to prevent some erroneous combinations. Further information on debugging features could not be explored as a publicly accessible version of BlockEditor could not be found.

3.1.13 Greenfoot. Greenfoot [P27] is a programming environment which uses a Java-like language, Stride. Stride uses a frame based system that uses blocks with editable text

sections. Selection of new blocks also occurs via keyboard entry. An animated transition is used to display the Java equivalent of any created block programs, but does not allow editing of the Java version. Stride removes many runtime errors by showing error messages to the user when erroneous input into any frame is detected. In addition, Stride features a full debugger, with line by line execution, and variable speed execution as well.

3.1.14 Pocket Code. Pocket Code [P30] is a mobile based BBPL quite similar to Scratch, in that it focuses on creating interactive games and other media based applications. Compared with App Inventor it is designed to create and run apps with no PC requirement. Compared to other BBPLs, Pocket Code has a limited number of blocks available, with even the almost universally present mathematics functions being absent.

For debugging, with the smaller selection of available blocks, Pocket Code actually becomes less prone to errors, as many common ones are simply not possible, which reduces the need for complex debugging. When an error does occur, Pocket Code ignores problem causing blocks. Pocket Code also features the ability to display co-ordinate axis for debugging the position of displayed items.

3.1.15 TouchDevelop. TouchDevelop [P31] is a BBPL designed specifically for touch devices and works on PCs, and mobile devices. The PC version also supports mouse and keyboard. TouchDevelop was originally defined as a text-based programming language that utilised an enforced structure to accommodate adding elements through touch events. However, since then a block view has been added and it can now be categorised as a BBPL. When using TouchDevelop users can decide to switch between three separate views, which are designed for different levels of programming proficiency, with the least experienced users being shown a block language, and the most experienced users being shown something very close to a textual language.

TouchDevelop constantly checks changed blocks and code, and shows in place error messages to make users aware of any issues. The program can still be run even when errors have been detected and doing so causes TouchDevelop to further highlight found errors, so users cannot ignore them. TouchDevelop is another language that features a line by line debugger, and also includes traditional debugging options such as step in, step out, step over, breakpoints, and even a simplified stack visualisation. However, these advanced options are only available in text mode.

3.2 Research Questions

3.2.1 RQ1 What Block Based Languages Exist? As can be seen in the section 3.1, a large number of BBPLs currently exist. After reviewing the papers it is clear that most existing languages can be categorised into two main groups: *introductory* and *transitional*, which roughly encapsulate the basic features of each. The few languages that do not fit, do not do so because they posses only some of the features required to elevate them into the transitional category, leading to the inclusion of a third category for these outliers: *mixed*. **Table 1** lists all BBPLs reviewed in this paper and their categorisation groups.

The first of these, the *introductory group*, consists of BBPLs that are primarily suited for introductory programming courses ranging from elementary to high school levels. Languages in this group generally lack more advanced programming features, but encourage new learners by easily allowing them to create fun and interesting programs through the use of blocks. One key aspect that is missing from BBPLs that fall under this category is the ability to view and edit creations in both block and text views. Without this users are eventually forced to move on to another environment to further pursue programming and research has shown that switching languages early on in the learning process can be detrimental [14]. However, some of the user studies performed in the reviewed papers show that new learners have still benefited from using such languages before moving onto text languages [P16].

The next category is the *transitional group*. Transitional languages possess a few key features which make them best suited for helping novice programmers at around the CS1 level transition into textual languages. Perhaps the most important of these features is the ability to switch between text and block based representations of a program, and additionally, the ability to edit either of these views and have the changes then be reflected in the other view. Another important feature is functional debugging. As novice programmers advance in knowledge, inevitably their programs become more complicated and thus a need for debugging features is practically inevitable. This is further backed by the knowledge that BBPLs often prevent many common errors such as syntax, which can often lead to an sudden influx of problems when novices are suddenly forced to consider these additional issues. Therefore, if novices start learning how to debug programs while still using blocks, they will have an easier time fixing errors in textual code. Lastly, transitional BBPLs need to possess advanced programming features. Similarly to debugging, as novice programmers progress they will likely end up using more complicated programming features such as recursion, multi-dimensional arrays, inheritance, etc. However, if a BBPL does not support these features, then novice learners are forced to move to a different language to use them, even if they are not quite ready to do so. While these three features are seen as being core features of transitional BBPLs, some BBPLs can be in the transitional group whilst only possessing some of these features in full. For example, Stride can be considered a transitional language despite not having full bidirectional text and block views, because it does feature advanced programming concepts and has full debugging capabilities. Conversely, Pencil Code

Table 1. BBPLs examined in this review.

Block Language	Text Language	Platform	Text Editor	Language Group
Scratch	None	PC	No	Introductory
App Inventor	None	PC, Android	No	Introductory
Pocket Code	None	Mobile	No	Introductory
Blockly	JS, Python, PHP, Lua, Dart	PC, Mobile	Unidirectional	Introductory
Calico Jigsaw	Python	PC	Unidirectional	Introductory
Alice	Java	PC	Unidirectional	Introductory
Pencil Code	CoffeeScript, JavaScript	PC	Bidirectional	Introductory
GP	LISP-like	PC	Bidirectional	Transitional
Madeup	MadeUp	PC	Bidirectional	Transitional
BlockPy	Python	PC	Bidirectional	Transitional
BlockEditor	Java	PC	Bidirectional	Transitional
BrickLayer	C	PC	Bidirectional	Transitional
Tiled Grace	Grace	PC	Bidirectional	Transitional
Stride	Java	PC	Unidirectional	Mixed
TouchDevelop	JavaScript	PC, Mobile	Bidirectional	Mixed

has been classified as a introductory language even though is has editable text and block views, because it does not possess advanced features and does not possess advanced debugging capabilities. Ultimately, the categorisation is not perfect, and further research could be done to truly evaluate this categorisation using a properly defined metric. However, the overarching theme is that these languages help novice learners move from a block based language to a point where moving onto a textual language is possible, with at most only a few potential difficulties.

3.2.2 RQ2 What Textual Programming Languages Do Block Based Languages Support?

BBPLs support a wide range of textual languages, though the means of support varies. Amongst the papers reviewed as part of this paper, the full list of supported languages is as follows: Java, Python, C, Grace, MadeUp, JavaScript, CoffeeScript, PHP, LUA, Dart, and Stride.

Out of these the most commonly occurring text language was Java and with Java having been the most common introductory computer science language for a number of years this is to be expected. Recent research has shown that Python has become the most common introductory language in US universities [5] and out of the BBPLs reviewed in this paper, Blockly, with export only, and BlockPy, with bidirectional, make use of Python. Another observation is the use of the Grace language, which is specifically designed for new learners, and is only used by a single BBPL. Few of the papers provide any reasoning behind why a specific text language was chosen, and questions remain unanswered as to the potential impact this choice of language could have on novice programmers. A graph showing the number of each BBPL using each textual programming language can be seen in **Figure 2**.

Table 2. Papers grouped by evaluation method.

Method	Papers
Survey	P8,16,26
Survey and Analysis	P2,10-11,13,20,24,27,33
CS1 Course	P35
CS1 Course and Analysis	P21,29
Analysis	P5,17,18,24,34
Observation	P9,12,19,23,31,32

3.2.3 RQ3 How Have Block Based Languages Been Evaluated?

BBPLs have been evaluated in a number of different ways. A summary of the evaluation methods, can be found in **Table 2**, while a more detailed overview of each reviewed paper with a user study can be found in **Appendix B**. While typical user studies consist of a survey, observation and analysis, many of the papers in this review only detail certain aspects of their user studies. The reported aspects are grouped using the following categories:

Survey includes all types of questionnaires and other forms of evaluation which involve questions being asked of the participants. This could be written, verbal or any other means of communication and includes all such interactions whether they be before, during or after any user studies. This also includes different types of survey methods, for example Likert scales [11], the Computer Attitude Scale [13] and Questionnaire Programming Knowledge [8].

CS1 Course refers to any user studies that were done as part of a first level undergraduate course. User studies that take place as part of a CS1 course can take place during part of, or all of the course, with evaluation methods consisting of quizzes and exams.

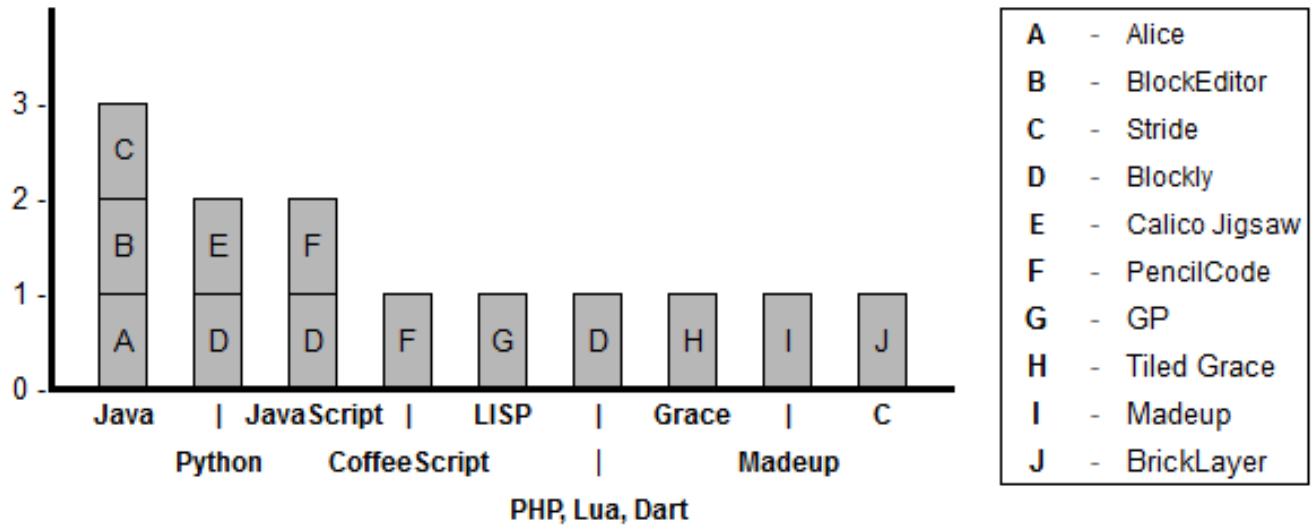

Figure 2. A graph showing the number of BBPLs that use each textual programming language. With the x-axis representing textual languages, and the key showing BBPLs.

Analysis is a broader category as it involves all evaluation methods that draw statistical conclusions from the data. In the examined papers this includes: evaluating how participants interacted with the software through a log or recording of performed actions as well as evaluating the outcome of the tasks performed. These tasks could be a specific user study task, or a task that took place as part of a CS1 course, for example.

Observation refers to papers that do not report any concrete results. While typical user studies contain an observation phase, this section is used to categorise papers that may not have done a real user study, but instead where data was gathered in a casual manner with no formal method being employed. Therefore, papers whose evaluation method falls into this category are unlikely to draw concrete conclusions and more likely to point towards areas with possible further research.

3.2.4 RQ4 What Educational Settings Have Block Based Languages Been Used in?

User studies for BBPLs have been performed with participants ranging from middle school to university level. The exact breakdown from the researched papers with user studies can be seen in **Table 3**. Including programming into elementary prospectuses is a recent change, and this is reflected in the user studies performed by the research papers, with no user studies including elementary students. However, Cheung et al. [P8] talk about summer workshops involving elementary students using Scratch, but provide no in-depth information or results from these events, leaving the possibility for further research to conduct user studies involving elementary students. **Table 3** also shows the total number of participants for each category, with University students being the clear winner at 1,114. Although there is a large disparity between the number of participants from each group, this is likely partly a reflection of the relative difficulty with which each group can be enlisted in user studies, with Universities simply having more students available in one place, and less hurdles to overcome for participation.

3.2.5 RQ5 What Computer Medium have Block Based Languages Been Deployed On?

The majority of BBPLs are designed to run in web browsers on PCs. This allows for a fair amount of platform independence and removes the need for downloading or installing any software, other than a web browser. This combination of factors removes potential hurdles and therefore makes a lot of sense for introducing BBPLs to new users. The results from the reviewed papers also show that a few BBPLs are designed for smartphones and tablets. While it can be more difficult to design BBPLs for such devices, due to smaller screens and different interaction methods, as mentioned previously the ubiquitous nature of mobile devices means that young people are more likely to have access to them, and therefore more likely to engage with BBPLs in their spare time.

Of the BBPLs reviewed in this paper, Pocket Code is the only one that is solely designed for mobile devices, however the features it possesses are quite limited even compared to other introductory BBPLs, limiting the experience that can be gained by novices using it. Another BBPL available on mobile devices is TouchDevelop, and there is a related comprehensive user study paper [10], testing its capabilities on mobile devices. However, this user study focuses more

Table 3. Papers grouped by education setting of user study.

Education Level	Papers	Participants	Highest
Middle School	P5,10,11,12,26,27,32,34	405	P32 (150)
High School	P2,4,8,19,20,24,31,33	423	P2 (120)
University	P9,13,16,17,18,21,23,29,35	1157	P21 (450)

on the non block based aspects of TouchDevelop, and was therefore not included in the reviewed papers. Lastly, Blockly is capable of being used on mobile devices, but requires use of a development tool to build and deploy apps that are made. This is similar to App Inventor, which is designed to deploy apps on Android devices, but the process for App Inventor seems to be more streamlined. This information points to an obvious need for more BBPLs on mobile devices and more related research, especially for BBPLs categorised under the introductory group intended for younger learners. Each BBPL examined in this review and the platforms it has been deployed on, is shown in **Table 1**.

3.2.6 RQ6 What kind of Debugging Features Do Block Based Languages Have?

Of all the user studies featured in the reviewed papers of this literature review, debugging is one of the least talked about aspects of BBPLs and very few research papers even mention it, with one notable exception being a user study conducted with Tiled Grace [P13], in the paper Homer and Noble showed clear evidence that participants of the user study found the debugging features to be beneficial, with the user study survey containing specific questions pertaining to them.

As debugging is a vital part of software creation, introducing debugging as part of introductory programming seems like a worthwhile venture and furthermore, the level of debugging capability is one indicator of whether a BBPL can be categorised as an introductory or transitional language. However, just because few of the user studies examined debugging, does not mean that the other BBPLs do not have debugging features. Many of them do, and section 2 contained information about the debugging features found from using all of the BBPLs that could be used, with the exceptions BrickLayer and BlockEditor having no publicly accessible version.

From the BBPLs examined in this review, a number of interesting debugging features were found. Chief amongst these is found in BlockPy, which has the ability to perform line-by-line execution in both forwards and backwards directions. The ability to debug backwards is a feature that is very rarely found in textual programming languages and as such really stands out. Another noteworthy feature is found in Tiled Grace and TouchDevelop, shown in **Figure 3**, which both show errors to the user in the place where it occurs as opposed to elsewhere on screen, which really helps novices to understand how their programs have gone wrong and where they need to apply fixes. In examining the debugging features of BBPLs it has also come to our attention that introductory BBPLs generally design their language to ignore errors and simply carry on executing. While this makes some sense, as novices may be unable to understand some error messages, it also can potentially prevent users from realising that their constructed program does not function as they intended, which is often a missed opportunity for further learning. Therefore, research that examines how exactly this type of error reporting impacts novices could be of use to the future BBPLs.

3.2.7 RQ7 What Kind of Influence Have Block Based Languages Had on Novice Programmers?

The results gathered as part of this paper clearly show that BBPLs are beneficial to people trying to learn programming. In particular, Malan and Leitner [P16] showed that 76% of participants in a Scratch user study felt that Scratch was a positive influence. While Papadakis et al. [P24] performed a user study with App Inventor, Scratch and Pascal, which showed through a series of tests related to a course, that participants using App Inventor and Scratch, showed greater improvements compared to the Pascal group, with the App Inventor group additionally performing better than the Scratch group. Price et al. [P27] also performed a user study comparing Stride and Java, and found that while levels of satisfaction and frustration was the same for both groups, participants using Stride completed tasks faster, and spent less time with erroneous code. Similarly, Price and Barnes [P26] performed a user study using a modified version of Tiled Grace, where participants were split into two groups, one restricted to the textual view and the other the block view. The results showed that participants using the block view, successfully completed more of the set tasks than the textual view group. Another interesting finding from Rizvi et al. [P29] was that students suspected of potentially having problems with CS1 courses, fared much better after taking an introductory CS0 course using BBPLs, and were more likely to subsequently take additional CS courses. Lastly, Wilson and Moffat [P34] showed that among the benefits of BBPLs, was that they can help prevent negative association with programming, which is especially important for younger learners.

In terms of BBPL groups, introductory BBPLs allow younger learners to learn by performing tasks that are relevant to their interests, such as games and animations, whilst also being easy to get into via exploratory programming. Whereas Transitional BBPLs have also been shown to be beneficial to older learners, as they allow for users to go at their own

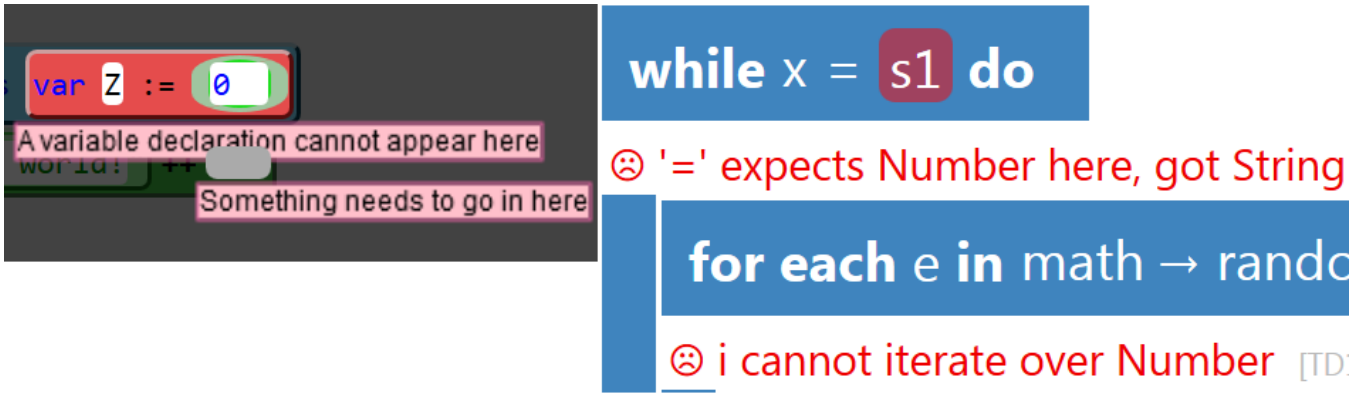

Figure 3. Multiple in-place error messages. Left: Tiled Grace, Right: TouchDevelop

pace, switching from blocks to text as necessary, in particular, Homer and Noble [P13] performed a user study in which participants found the two view system to be enjoyable. User studies of transitional BBPLs also showed that users move away from block views as they improve, eventually leading to almost exclusive usage of textual view as they develop their skills, of these Bau et al. [P5] showed participants reach 95% text view usage over a four session user study, while Matsuzawa et al. [P18] found that the rate of block usage in their user study dropped to around 10% over the course of a 15 week study.

There were also some user studies that highlighted problem areas, for example, Mishra et al. [P21] showed that while Scratch was useful for novice learners, in that it improved their performance with subsequent C++ questions, it did not sufficiently improve performance in C++ debugging questions. Meerbaum-Salant et al. [P19] also highlight some bad habits that novices can learn from Scratch, including a bottom-up programming approach in which participants would drag all seemingly relevant blocks for solving a task onto the canvas, and then try to guess how they best fit together, rather than thinking about the task in terms of algorithms or software design. Participants also were found to use extremely fine-grained programming, where they would decompose tasks into needlessly small subtasks. Weintrop and Wilensky [P33] identified a number of potential problems of BBPLs, including the feeling that BBPLs were not real languages, being unable to express all concepts, and large programs being hard to manage. These points are especially interesting as the paper's user study was done with Snap! [6], an extension for Scratch which adds greater functionality, as Snap! is designed to better suit the needs of more advanced learners, yet participants still found it to be too limiting. Lastly, some papers found that Scratch did not provide novices with sufficient understanding of certain concepts, including initialisation, concurrency, variables, loops and booleans, [P20, P11] despite the user study participants having little trouble creating programs.

4 Conclusions

A number of interesting points have arisen as part of this literature review. Firstly, the categorisation potential of most BBPLs into the Introductory and Transitional groups. Introductory BBPLs focus on enabling younger users to create fun games and animations via blocks only. Transitional BBPLs posses block and text views that allow users to see how blocks transition into actual code, as well as more advanced debugging and programming features that ultimately help users move on from blocks to textual coding. It was also observed that some languages can be difficult to categorise as they may feature only a subset of the transitional language criteria, with TouchDevelop and Stride being notable examples that warrant the inclusion of an additional Mixed category. Even with these two outliers We believe that this categorisation has merit, as it can be used to guide the selection the ideal language for a given age group.

The textual language used by those BBPLs that have a textual view is also worthy of investigation. There exists a wide selection of text languages in use by BBPLs. From the BBPLs reviewed in this paper, Java and Python are the most popular choices with three and two respective uses, see **Figure 2**. However little information is given as to why a specific language was chosen and questions remain as to whether BBPLs are possibly better for certain languages or whether they need to be adjusted to help transition into certain languages. Additionally, whether languages that are the most popular currently, such as Python or Java, or languages specifically designed for new learners, such as Grace, are better choices for BBPLs to use.

Examining the environments that BBPLs have been evaluated in shows that while introductory undergraduate courses, high schools, and middle schools, have had a number of user studies conducted in them, elementary schools have not. With recent changes to school curriculums around the world bringing the age at which students start to learn programming lower, investigating how younger learners may benefit from BBPLs is critical, and Bell et al. [1] highlight the need

for new computer science courses aimed at younger learners to be carefully designed to ensure that learners are aware of potential career paths and a sufficient range of topics are available to match the abilities of students. In terms of BBPLs, this can be viewed as ensuring that students know why they are learning programming, while they learn how to program. Additionally, ensuring that students are using a BBPL that matches their age and ability range is paramount, hence the need to further evaluate this topic.

As far as platform goes, the vast majority of BBPLs are designed for deployment in PC web browsers. Viewing such web pages on a mobile device is possible, but may lead to difficulties due to the smaller screen space and differing input methods. Providing a dedicated mobile application or mobile friendly web version would lessen the severity of these issues, which is especially important when dealing with younger learners who may be both less tolerant of such problems, and less able to overcome them.

A Reviewed Papers

References

[P1] Saksham Aggarwal, David Anthony Bau, and David Bau. "A blocks-based editor for HTML code". In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE. 2015, pp. 83–85.

[P2] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. "From Scratch to "real" programming". In: *ACM Transactions on Computing Education (TOCE)* 14.4 (2015), p. 25.

[P3] Austin Cory Bart et al. "Position paper: From interest to usefulness with BlockPy, a block-based, educational environment". In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE. 2015, pp. 87–89.

[P4] David Bau. "Droplet, a blocks-based editor for text code". In: *Journal of Computing Sciences in Colleges* 30.6 (2015), pp. 138–144.

[P5] David Bau et al. "Pencil code: block code for a text world". In: *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM. 2015, pp. 445–448.

[P6] Douglas Blank et al. "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*. ACM. 2012, pp. 63–68.

[P7] Karishma Chadha and Franklyn A Turbak. "Improving App Inventor Usability via Conversion between Blocks and Text." In: *J. Vis. Lang. Comput.* 25.6 (2014), pp. 1042–1043.

[P8] Joey CY Cheung et al. "Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students". In: *ACM SIGCSE Bulletin*. Vol. 41. 1. ACM. 2009, pp. 276–280.

[P9] Matthew Conway et al. "Alice: lessons learned from building a 3D system for novices". In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. 2000, pp. 486–493.

[P10] Hilary Dwyer et al. "Fourth Grade Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances". In: *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM. 2015, pp. 111–119.

[P11] Shuchi Grover and Satabdi Basu. "Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM. 2017, pp. 267–272.

[P12] Shuchi Grover and Roy Pea. "Using a discourse- intensive pedagogy and Android's App Inventor for introducing computational concepts to middle school students". In: *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM. 2013, pp. 723–728.

[P13] Michael Homer and James Noble. "Combining tiled and textual views of code". In: *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE. 2014, pp. 1–10.

[P14] Chris Johnson and Peter Bui. "Blocks in, blocks out: A language for 3D models". In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE. 2015, pp. 77–82.

[P15] Michael Kölling, Neil CC Brown, and Amjad Altadmri. "Frame-based editing: Easing the transition from blocks to text-based programming". In: *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM. 2015, pp. 29–38.

[P16] David J Malan and Henry H Leitner. "Scratch for budding computer scientists". In: *ACM SIGCSE Bulletin* 39.1 (2007), pp. 223–227.

[P17] Yoshiaki Matsuzawa, Yoshiki Tanaka, and Sanshiro Sakai. "Measuring an Impact of Block-Based Language in Introductory Programming". In: *International Conference on Stakeholders and Information Technology in Education*. Springer. 2016, pp. 16–25.

[P18] Yoshiaki Matsuzawa et al. "Language migration in non-CS introductory programming through mutual language translation environment". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM. 2015, pp. 185–190.

[P19] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. "Habits of programming in Scratch". In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM. 2011, pp. 168–172.

[P20] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. "Learning computer science concepts with Scratch". In: *Computer Science Education* 23.3 (2013), pp. 239–264.

[P21] Shitanshu Mishra et al. "Effect of a 2-week Scratch intervention in CS1 on learners with varying prior knowledge". In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM. 2014, pp. 45–50.

[P22] Jens Monig, Yoshiki Ohshima, and John Maloney. "Blocks at your fingertips: Blurring the line between blocks and text in GP". In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE. 2015, pp. 51–53.

[P23] Ralph Morelli et al. "Can Android App Inventor bring computational thinking to k-12". In: *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)*. 2011, pp. 1–6.

[P24] Stamatios Papadakis et al. "Using Scratch and App Inventor for teaching introductory programming in secondary education. A case study". In: *International Journal of Technology Enhanced Learning* 8.3-4 (2016), pp. 217–233.

[P25] Shaileen Crawford Pokress and José Juan Dominguez Veiga. "MIT App Inventor: Enabling personal mobile computing". In: *arXiv preprint arXiv:1310.2830* (2013).

[P26] Thomas W Price and Tiffany Barnes. "Comparing textual and block interfaces in a novice programming environment". In: *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM. 2015, pp. 91–99.

[P27] Thomas W Price et al. "Evaluation of a Frame-based Programming Editor." In: *ICER*. 2016, pp. 33–42.

[P28] Mitchel Resnick et al. "Scratch: programming for all". In: *Communications of the ACM* 52.11 (2009), pp. 60–67.

[P29] Mona Rizvi et al. "A CS0 course using scratch". In: *Journal of Computing Sciences in Colleges* 26.3 (2011), pp. 19–27.

[P30] Wolfgang Slany. "Tinkering with Pocket Code, a Scratch-like programming app for your smartphone". In: *Proc. of Constructionism* (2014).

[P31] Nikolai Tillmann et al. "TouchDevelop: programming cloud-connected mobile devices via touchscreen". In: *Proceedings of the 10th SIGPLAN symposium on New ideas, New Paradigms, and Reflections on Programming and Software*. ACM. 2011, pp. 49–60.

[P32] Barbara Walters and Vicki Jones. "Middle school experience with visual programming environments". In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE. 2015, pp. 133–137.

[P33] David Weintrop and Uri 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*. ACM. 2015, pp. 199–208.

[P34] Amanda Wilson and David C Moffat. "Evaluating Scratch to introduce younger schoolchildren to programming". In: *Proceedings of the 22nd Annual Psychology of Programming Interest Group (Universidad Carlos III de Madrid, Leganés, Spain* (2010).

[P35] David Wolber. "App Inventor and real-world motivation". In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM. 2011, pp. 601–606.

B Reviewed Papers with User Studies

Table 4. Information from each reviewed paper with a user study component.

Paper	Language	Education	# Part.	Eval. Method
P2	Scratch	High School	120	Survey,Analysis
P4	Droplet (PencilCode)	High School	14	Survey
P5	App Inventor	High School	8	Analysis
P8	BrickLayer	Middle School High School	24	Survey
P9	Alice	University	100	Observation
P10	Scratch	Middle School	26	Survey, Analysis
P11	Scratch	Middle School	100	Survey,Analysis
P12	App Inventor	Middle School	7	Observation
P13	Tiled Grace	University	33	Survey, Analysis
P16	Scratch	University	25	Survey
P17	BlockEditor	University	100	CS1
P18	BlockEditor	University	404	Analysis
P19	Scratch	Middle School	46	Observation
P20	Scratch	High School	56	Survey,Analysis
P21	Scratch	University	450	CS1
P23	App Inventor	University	2	Observation
P24	App Inventor Scratch	High School	87	Survey, Analysis
P27	Greenfoot	Middle School	24	Survey,Analysis
P29	Scratch	University	43	CS1,Analysis
P31	Pocket Code	High School	24	Observation
P32	App Inventor	Middle School	150	Observation
P33	Snap!	High School	90	Survey,Analysis
P34	Scratch	Middle School	21	Analysis
P35	App Inventor	University	-	Observation

References

[1] Tim Bell, Peter Andreae, and Lynn Lambert. "Computer science in New Zealand high schools". In: *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*. Australian Computer Society, Inc. 2010, pp. 15–22.

[2] Andrew P Black et al. "Grace: the absence of (inessential) difficulty". In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM. 2012, pp. 85–98.

[3] Wanda P Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice (w/CD ROM)*. Prentice Hall Press, 2011.

[4] Carl Benedikt Frey and Michael A Osborne. "The future of employment: how susceptible are jobs to computerisation?" In: *Technological Forecasting and Social Change* 114 (2017), pp. 254–280.

[5] Philip Guo. *Python is now the most popular introductory teaching language at top US universities*. https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext. Accessed: 2017-07-3.

[6] Brian Harvey et al. "Snap!(build your own blocks)". In: *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM. 2014, pp. 749–749.

[7] Staffs Keele et al. "Guidelines for performing systematic literature reviews in software engineering". In: *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn, 2007.

[8] Sebastian Kleinschmager and Stefan Hanenberg. "How to rate programming skills in programming experiments?: a preliminary, exploratory, study based on university marks, pretests, and self-estimation". In: *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*. ACM. 2011, pp. 15–24.

[9] US Bureau of Labor Statistics. *Occupational Outlook Handbook - Computer and Information Research Scientists*. https://www.bls.gov/ooh/computer-and-information-technology/computer-and-information-research-scientists.htm. Accessed: 2017-07-02.

[10] Sihan Li, Tao Xie, and Nikolai Tillmann. "A comprehensive field study of end-user programming on mobile devices". In: *Visual languages and human-centric computing (vl/hcc), 2013 ieee symposium on*. IEEE. 2013, pp. 43–50.

[11] Rensis Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932).

[12] Jesús Moreno-León and Gregorio Robles. "Code to learn with Scratch? A systematic literature review". In: *Global Engineering Education Conference (EDUCON), 2016*. IEEE. 2016, pp. 150–156.

[13] Gary S Nickell and John N Pinto. "The computer attitude scale". In: *Computers in human behavior* 2.4 (1986), pp. 301–306.

[14] Kris Powers, Stacey Ecott, and Leanne M Hirshfield. "Through the looking glass: teaching CS0 with Alice". In: *ACM SIGCSE Bulletin* 39.1 (2007), pp. 213–217.

[15] Mitchel Resnick. "Point of view: Reviving papert's dream". In: *Educational Technology* 52.4 (2012), p. 42.

[16] Taylor Soper. *Analysis: The exploding demand for computer science education, and why America needs to keep up*. https://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/. Accessed: 2017-07-01.

[17] *TIOBE Index | Top 20 Programming Languages July 2017*. https://www.tiobe.com/tiobe-index/. Accessed: 2017-07-05. 2017.

[18] Franklyn Turbak et al. "2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)". In: *Atlanta* 2015 (2015).

[19] Department for Education UK Government. *National curriculum in England: computing programmes of study*. https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study. Accessed: 2017-07-10.

Received 2022-09-01; accepted 2022-10-02