

Using Virtual World Programming Languages To Teach Computer Science Concepts

Brett Ward

November 5, 2009

Department of Computer Science & Software Engineering

University of Canterbury, Christchurch, New Zealand

Supervisor: Tim Bell

Contents:

1. Abstract	3
2. Introduction	4
3. Design Study Overview	6
3.1. Programming Languages for School Students	6
3.2. A Look at the Chosen Concepts.....	6
3.2.1. Sorting Network.....	7
3.2.2. Sorting Algorithms	7
3.2.3. Binary Number Operations	8
3.3. Design Study Layout.....	8
4. Alice	9
4.1. The Concepts in Alice	10
4.1.1. Sorting Network	11
4.1.2. Sorting Algorithms	12
4.1.3. Binary Number Operations	13
4.2. Additional Concepts Attempted	13
4.3. Features and Issues	14
4.4. Discussion	18
5. Scratch	20
5.1. The Concepts in Scratch	21
5.1.1. Sorting Network	21
5.1.2. Sorting Algorithms	22
5.1.3. Binary Number Operations	22
5.2. Features and Issues	23
5.3. Discussion	24
6. Greenfoot	26
6.1. The Concepts in Greenfoot	26
6.1.1. Sorting Network	27
6.1.2. Sorting Algorithms	27
6.1.3. Binary Number Operations	28
6.2. Features and Issues	29
6.3. Discussion.....	30
7. Future Work	31
7.1. Other Promising Languages	31
8. Conclusions	33
9. References	35
10. Appendices	37

1. Abstract

With many secondary-level curricula being updated to incorporate a larger amount of computer science concepts, there is a need to identify sufficient ways to teach these concepts within languages commonly used at the appropriate age levels. Currently, languages like Alice, Scratch and Greenfoot, among many others, are both freely available and widely used to teach aspects such as programming, but little research has been done on whether they can actually be used to easily and sufficiently teach other concepts, such as algorithms and data representation. This paper discusses these such languages, and takes a look at how usable they actually are for performing some simple tasks. A number of computer science concepts are looked at in these languages, with implementation possibilities and difficulties overviewed, and discussion on how these languages could be enhanced to make it easier to teach the chosen concepts within them.

2. Introduction

Education in Computer Science (CS) is currently a large and rather mixed topic. Many secondary schools focus purely on teaching programming, with a varying array of languages such as Alice, Scratch, Java and Python being used, while others additionally teach some other concepts. Many schools also neglect to provide any sort of computer science course. Surveys like those done by the CSTA (<http://www.csta.acm.org/>) show these also show how much the topics covered can vary between schools [4]. But there are signs of change, with many curricula, such as the K-12 guidelines [14] and countries such as NZ (with a public draft available on <http://www.techlink.org.nz/curriculum-support/tks/>, as well discussion about the curriculum in [2]), the UK, and the USA being altered or updated to focus more on concepts related to computer science that are not directly programming in addition to the traditional programming skills taught.

What CS actually is can often become also a rather confusing topic. Wikipedia (<http://en.wikipedia.org/>) explains CS as “the study of the theoretical foundations of information and computation”, with a large number of sub-fields including computational theory, computer graphics, programming language theory, and many more. The K-12 guidelines [14] describe CS as “the study of computers and algorithmic processes, including their principles, their hardware and software designs, their applications, and their impact on society.” In these cases the theory, concepts and principles are separated from the programming aspects. CS can also be clouded by how computers are used within schools – as tools for learning or creating documents, spreadsheets, or similar general purpose applications, which sometimes aids in this confusion. This is also discussed in [2].

CS concepts are also often difficult to define. One way to look at these, using an example similar to that found in [2], as well as ideas raised in the NZ curriculum, is as a difference between designing and construction. Designing requires understanding how a concept works, how data structures used by the concept work, and similar ideas, while constructing requires knowledge of programming languages and concepts for creating the design in code. These ideas, such as algorithms, data representation, networks and routing, and interface design are an important part of computer science, but are often lumped together with programming concepts such as loops, expressions, or more complex ideas like object-oriented languages.

The K-12 guidelines also explore a grade 6-8 course with the intention that, at grade 8, students can “Demonstrate an understanding of concepts underlying hardware, software, algorithms, and their practical applications”. An introduction into programming is not recommended until later years, with more advanced topics such as recursion and event-driven programming not being included until courses several years after this. The upcoming NZ curriculum draft is also similar, separating concepts of algorithms from the concepts of programming in early years, with Level 6 (Year 11, previously Form 5) topics covering the concepts of algorithms and their costs, but only touching on basic programming such as loops and expressions. Because of these, not only the ability to teach computer science early is needed, but a way to implement solutions when relevant programming concepts are not

yet taught.

Initiatives such as CSUnplugged (<http://csunplugged.org/>), CS4FN (<http://www.cs4fn.org/>) and CSInside (<http://csi.dcs.gla.ac.uk/>) aim to provide resources to help teach these CS concepts in numerous ways, generally without the need for programming. CSUnplugged, for example, aims to teach concepts such as those discussed in this paper without the need for a computer by providing activities that can be done anywhere from on a whiteboard or with a pen and paper to activities that a large group can participate in using a large outdoor area. Of course, initiatives like these have their own limitations – for example there is not always enough space to draw out a sorting network available. At times these concepts can be shown easily graphically using a computer, but many languages require complex graphical code to be learnt in addition to the code that would normally be used to implement these concepts, making teaching through having students implement ideas and concepts difficult.

This is where visual programming languages could be used. Languages like Alice are often used to teach programming at early stages. They have been found to increase retention, allow some aspects of programming to be more easily taught, and of course tend to be more highly enjoyed by the users as they can see what they're doing [11,5]. But currently there has been very little work in relation to teaching concepts, like those used by CSUnplugged, in these languages, although a number of ideas on what may be applicable have been produced in papers previously with little or no advice on how they would be implemented [12].

Languages and virtual worlds such as Second Life (<http://secondlife.com/>) or OpenSim (<http://opensimulator.org/>) have also been used for educational purposes. These contain scripting languages that can allow a person, or groups of people from various locations, to interact with a predefined setup such as a sorting network. Websites like SLENZ (<http://slenz.wordpress.com/>) and the Second Life Education Wiki (<http://sleducation.wikispaces.com/>) explore these educational uses of both Second Life and other online virtual worlds. Whilst languages like these are not the focus in this paper, it is important to acknowledge the fact that large scale virtual worlds such as these can and have been used to teach educational concepts.

This report takes an in-depth look at a number of these visual programming languages and how a few particular computer science concepts could possibly be implemented, in some cases not easily, in them. First, the concepts and languages chosen are outlined, before a discussion of how each language performed in trying to fulfil these concepts. Within each section, problems specific to that language are also discussed. To finish, some discussion about languages not investigated here is followed by some conclusions on how suited each language performed compared to the others.

3. Design Study Overview

With a wide range of both languages to use, and concepts to implement, a decision on which choices would be most practical needed to be made. To do this, a basic look into what languages were currently being used, and what concepts would be worthwhile to try given upcoming curricula changes was done, and these facts were used to help choose the components of this study. A short overview of these decisions follows.

3.1. Programming Languages for School Students

With a very wide range of programming languages available, and an increasing number of these providing the tools to create virtual worlds, it was important to pick languages that were appropriate to the content and age levels of those who would be using the language within these new curricula – typically teenagers, but also looking at users outside this bracket.

Other factors such as availability, usage inside schools, other community activity (such as discussion groups), and any required costs or constraints of use with the language were considered. Because of this, the languages Alice 2.2 (<http://alice.org/>), from Carnegie Mellon University, Scratch (<http://scratch.mit.edu/>), from MIT's Lifelong Kindergarten Group, and Greenfoot (<http://www.greenfoot.org/>), from Michael Kolling and the University of Kent were chosen. Other languages such as LOGO, Kodu, OpenSim and Second Life were also looked at as possibilities, as well as a newer version of Alice that was in beta during the time of this report, but not chosen due to factors discussed later in this document

All three of these chosen languages are freely available from their given websites, although versions newer than those stated in the relevant sections may be available in some cases. Versions used and other information can be found in the section related to each language.

Each of these languages provides an implementation of a virtual world in which the programmer can see their program come to life. Whilst Alice is in 3D and Scratch and Greenfoot represent 2D virtual worlds, the comparison between 2D and 3D was not a direct part of this study. Each language also provides some sort of export option, allowing projects to be shared in various forms.

With each language not only was their ability to show the given concepts explored, but investigation into their general usage, including problems such as general program stability, bugs, awkwardly implemented features and missing programming constructs, was also completed.

3.2. Concepts Chosen

Three major concepts were chosen and attempted to be implemented in each of the above languages, each aiming to provide a different type of implementation strategy, different use of programming constructs and with a hope that they would find different issues within each virtual world.

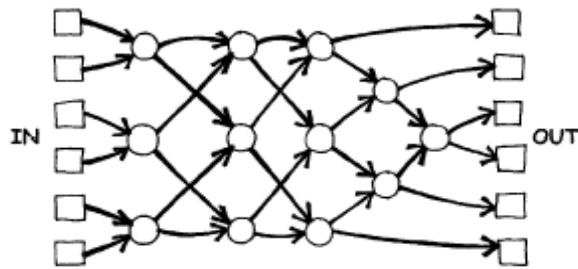


Figure 1: A simple sorting network example, as used by the CSUnplugged "Sorting Networks" activity

The first concept chosen was that of a sorting network, the second sorting algorithms, and the third a look at binary number representation and conversion. It was found that performing binary number calculations in various forms was easy to show in these languages, and that Scratch already provided examples through projects available online. In this case the idea of using binary was expanded to include parity error checking, although possible implementation methods of a binary calculator were also investigated.

In each case the aim was to find possible ways to implement these ideas, with the hope that these implementations would be similar to those of the actual algorithms or concepts being shown in non-visual languages such as Java. A brief description of each task follows, as well as the reasoning of each choice.

3.2.1. *Sorting Networks*

Sorting networks, such as Figure 1, can be used to show computational concepts such as threaded processing and object comparison. They also have the benefit of being highly visual, as each network can be shown as a series of nodes and paths, with comparisons happening at each point where items meet.

Each object within the network is given some information which can be compared, in this case simple numbers will just be used, and when two objects reach the same part of the network, such as a circle in Figure 1 they are compared, moving to the next zone appropriate to the result of the comparison. When all objects reach the end of the network they are in sorted order.

This concept was chosen because not only was it highly visual, but it required consideration of aspects such as movement within the virtual world, placing objects, and of course the concept of comparison. This allowed each virtual world to be investigated for any issues that may arise when creating a world that required more simplistic programming, but a higher level of virtual world manipulation.

3.2.2. *Sorting Algorithms*

Sorting algorithms both provide a wide range of programming skills to be tested, and show concepts such as recursion and how different algorithms perform the same task at different speeds using different strategies. In each

case, a basic selection sort was created. In Alice a quicksort was also created, and although this was not completed in Greenfoot it is likely possible given the power available to it.

With this concept, the ability of a virtual world to produce a large array was tested. This led to a number of features of the virtual world being put to the test, such as the ability to show large numbers of objects clearly and effectively, how well the virtual world handled a significant number of items, and how easily an array could be visualised in the world, with updates to the array also being shown easily.

Because each of the virtual worlds being tested gave the ability to change the scale of an object, this feature was used as the variable to sort by. Heights of objects were manipulated, whether through choosing characters of different heights or selecting similar characters and altering the height variables.

3.2.3. *Binary Numbers*

The concept of binary numbers is a key part of computer science. While binary numbers can themselves be a rather primitive concept, performing binary calculations and being able to read binary numbers are key skills in some computer science fields - for example, binary representations are important for information theory, priority queues (binomial heaps), numerical methods, data compression, cryptography, and image representation (colour models).. A number of different ideas were implemented for these concepts, but in each case a basic binary calculator with mouse events was created.

In this concept, the event binding capability and simplicity was a major factor. In languages such as Java dealing with event bindings and listeners can be quite a complicated feature, therefore reducing the need for students with limited programming knowledge to implement significant event code was an important aspect to be considered.

3.3. *Layout of Study*

The form of the study is laid out based on each language. Initially, some background into the language is discussed, giving an overview of what each interface looks like, some discussion on how the software works, and noting relevant papers reinforcing or disputing the ideas and features given by these design choices.

From there, each concept is discussed, with many of the important features brought forth through this study outlined. Images are used to help show both the differences in style between each language, and reflect on features that are easily visible in the virtual world.

Finally, an overview of other features of the language not previously mentioned is discussed, including how the export feature works in each system, and various benefits and limitations provided by the features unique to it. This is followed by some discussion on overall performance.

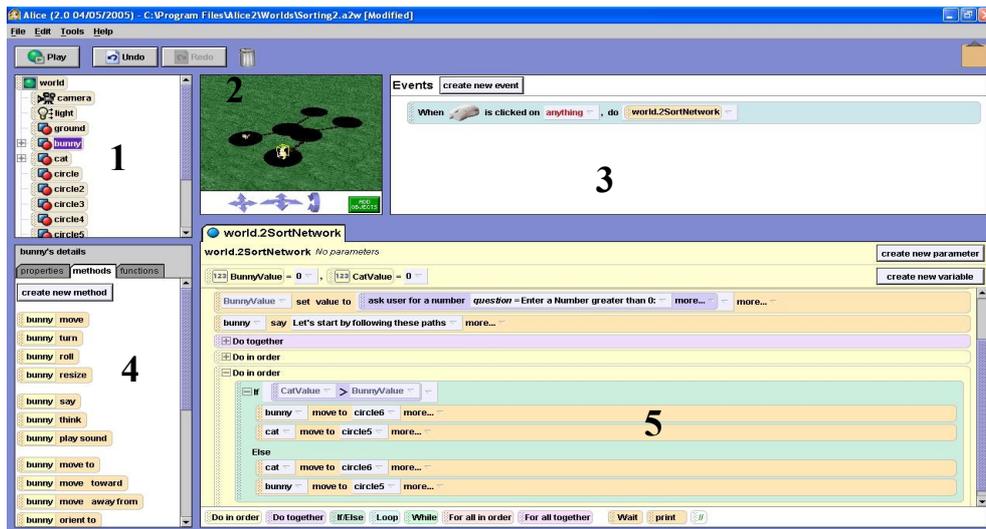


Figure 2: The Alice interface, split into 5 distinct parts. 1) The object tree, where all objects within the virtual world reside. 2) The virtual world. 3) Event bindings tied to the world. 4) Code blocks for the selected object. 5) The method currently being edited

4. Alice

First released in the late 90s, Alice has become quite a popular beginner programming language, with over 1800 schools being listed as using Alice on the Alice homepage. It provides a visual programming language with the aim to aid in giving people a taste of object-oriented programming. Designed as a “first exposure” language, it gives many of the simple constructs and programming concepts used in programming languages, such as arrays, recursion, and a number of variable types, and allows for characters and objects in a virtual world to easily be programmed to create worlds of varying complexity. It is aimed at children at around the high school level, although some institutions use it as an introductory programming language at university level before moving on to more complex languages like Java.

Alice uses a “drag and drop” style interface, seen in Figure 2, with part 4 showing a number of draggable code blocks, which when moved into the program editor (Figure 2, part 5) produce a new statement in the code. Code is never typed in Alice, except for the naming of variables and some awkward pre-defined functions that require it, instead everything is dragged and dropped in this style. It also provides a rich yet simple event binding system (Figure 2, part 3), allowing students to create complex mouse and keyboard bindings with a simple drag and drop, minimising the need to teach an often complex part of languages. Alice also features a general list of objects in the world, shown in Figure 2 (part 1) as well as the world itself (Figure 2, part 2).

A lot of research has previously been done on how Alice is good at teaching programming, whilst having a number of flaws. Papers such as [3] discuss a number of “good and not so good” aspects of Alice, some of which are also mentioned in [11], which also discusses how Alice has been shown to increase retention in early CS courses. Similar papers, such as [9], also discuss not only problems with how going from Alice to Java can be a

problem, but ideas for minimizing this. Many of the ideas touched on in these papers are also reflected upon in this study. Using Alice to teach recursion was a topic discussed from original versions [6], and still very appropriate in current versions. There are also a number of books that aim to teach introductory programming in Alice, such as [7], although the topics raised in these focus less on teaching concepts and more on teaching programming, often using storytelling as a means of motivation

Currently there are a number of readily downloadable versions of Alice – versions 2, 2.2, and a beta of version 3. All programs created in this design study were done so in version 2.2 (build dated 30/4/2009), as this was the most stable version during the time of this report. Version 3 was also assessed to see if it could be used in the currently available form, but no significant programs were made. Notes on the Alice 3 beta are mentioned later in this document.

4.1. The Concepts in Alice

Each of the concepts was successfully implemented within Alice, although investigation into ways to implement some of these concepts was needed before suitable implementations were found. However, while creating these a significant number of potential problems were found within Alice, such as crash-causing bugs, limitations of the software, and difficulties in interacting with various parts of the system. A number of these issues are discussed in detail in the next section. This is not to say Alice is a terrible system to use, as in fact it is quite the opposite, and a number of positive features have also been mentioned, as well as ideas for avoiding a number of the more problematic issues.

Additionally, as Alice was the first language studied, other concepts were investigated for different possible implementation styles. Some comments have been made on these concepts and decisions as to why they were not completed as part of the design study.

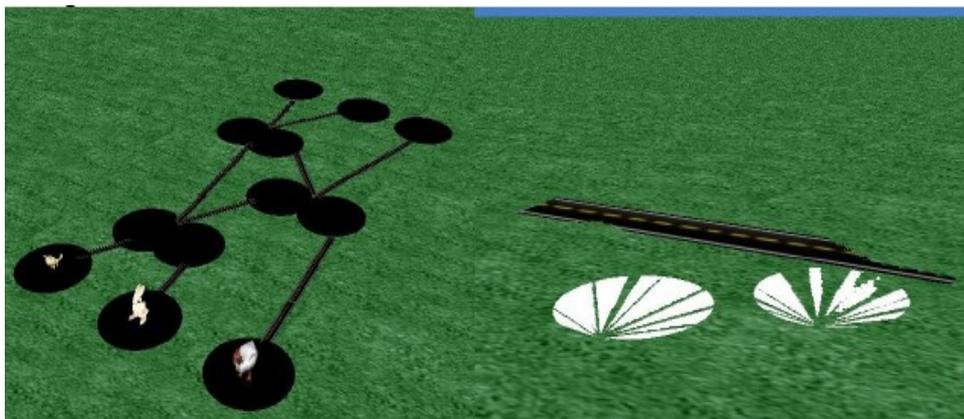


Figure 3: (left) A simple network shown in Alice using circles as nodes and roads as paths. (right) These flat objects show how the ground surface in Alice is not flat



Figure 4: This function allows variables not predefined by Alice to be referenced in a generic fashion, but also causes encapsulation issues

4.1.1. Sorting Network

To begin with, we attempted to try a basic three variable sorting network. The initial aim was to create an automated network where the variables being sorted could be changed, and the network still complete the operation correctly. It was found this task was rather difficult, as Alice lacked the ability to compare objects in the virtual world by variables such as the distance between the characters, or by recognising where they were in the world.

Because of these issues, a “hard-wired” version was initially implemented. This version simply animated the movements of a specific case in the sorting network. This version was rather simple to show in Alice, as the sorting network could be placed on the surface using objects such as squares, circles, and in this case roads as the paths between nodes, and with careful use of the movement functions such as “move to” to move a character from one node to another. An implementation of this can be seen in Figure 3 (left). In this case, the general principle of how the sorting network works would still need to be understood by the student to complete the task of sorting the objects successfully.

As was also quickly discovered, manually placing the objects on the surface provided by Alice was tedious, and it was quickly found that placing an object, such as the flat squares or circles used, at ground level often made parts of the object disappear, as if the surface was not flat. This is shown in Figure 3 (right). To get around this problem, all objects were simply placed a small amount above ground level, although this added some additional work in placement, and caused objects to look like they were floating unnaturally at certain camera angles. Another issue, when using pre-defined methods such as “move to” to move a character from their current position to a new object, some objects moved in two dimensions, others moved in three, making it difficult to program for a generic movement case. This issue can be resolved by carefully selecting which objects are used, and it was noted that objects that had models that consisted of multiple parts, such as a horse, were more likely to move in 3 dimensions, where a single part model, such as that of a chicken, would only move in 2 dimensions.

After further investigation into Alice, a special type of “visualisation” objects were found. These objects allowed nodes in the network structure to be replaced with arrays, where the two objects within the array could easily be compared at each point without knowing which object was specific to each array position. This allowed the steps in the array to be awkwardly automated, but increased the level of complexity – simple arrays holding two objects is unnecessarily complex. The use of these visualisations had other advantages, though – in particular while using them the need for movement coding was significantly reduced, as the visualisation objects provide methods that allow the array changes, both visual and textual, to be done within one code block.

Another issue encountered while creating this concept was the difficulty in using an objects existing variables as items to compare in the sort. Every object in an Alice world has variables such as “height” and “width”, which

can easily be compared between two objects, but if a user wishes to give a new unique variable to every object, for example a number to sort them by, Alice will not easily allow these variables to be accessed between each object – even if all the objects are of the same type, because Alice represents each object as a individual object, rather than using a class-oriented methodology. However, it does provide an awkward way to reference such a variable through the function shown in Figure 4. If each variable is given the same name, it can be identified by using this function and then used similar to a generic class method or variable. This forces the user to assume that all objects have this given variable, which requires a compromise in encapsulation.

4.1.2. *Sorting Algorithms*

Of the three concepts focused on, Alice showed the most promise here due to a pre-defined object called the “Array Visualisation”, shown in Figure 5 with a number of characters sitting on it, and mentioned briefly in the previous concept. This object provides the ability to create an array much like those used in other languages such as Java, and perform both operations on an array and the visual aspects of the array change through simple pre-defined methods. With the need to program the visualisation aspects of the world minimized, both the selection sort and quicksort algorithms could be made remarkably close to that of those taught in existing courses. The visualisation also simplifies putting characters into place in the world, as when a character is added to an array slot through the code editor Alice automatically moves it into the correct slot in the virtual world.

With use of this array visualisation, arrays such as those above were rather easy to create, but creating a larger array proved difficult – the array visualisation has a set size per section, and it is rather difficult to see objects in the virtual world from a distance, making a full overview of a large array difficult to visualise.

With these factors, a simple 8 object array was created. An object visualisation was also used to sit an object on as while the other object was moved to the slot it previously occupied – much like using a temporary variable in a “swap” method. From here, the code was rather simple, as existing implementations of the sorting algorithms could essentially be built in Alice using the expected combination of code blocks.



Figure 5: The "Array Visualisation" object allows sorting algorithms to be implemented rather easily



Figure 6: A simple binary calculator in Alice

4.1.3. Binary Number Operations

The event binding in Alice makes concepts such as binary conversion and parity rather simple to create in a visual manner – in this case the use of the “when I click anything” event removes any significant coding that would be required in other languages. From here, the “anything” object can be passed as a parameter into the required code. There are many methods for determining if an object requires an action when clicked, but in this instance having each bit in an array and iterating through that array to confirm if the object clicked was indeed a bit proved the simplest method. Alternate methods such as giving each bit a unique click event quickly cluttered up the small event binding code area. Alice, however, has one small but avoidable issue with using the “when I click anything” event – clicking the sky of the virtual world will cause Alice to throw an exception error. No way to code around this was found, as the sky is not represented as an object in the Alice world.

In this implementation, shown in Figure 6, a box was used to represent a bit, with a simple colour swap (from white to black) occurring if a bit was clicked. The text used is a “3D Text” object in Alice, which is easily updated whenever a bit is changed.

4.2. Other Concepts Attempted In Alice

The concept of router deadlock was also explored, with the hope that an activity similar to that used by CSUnplugged

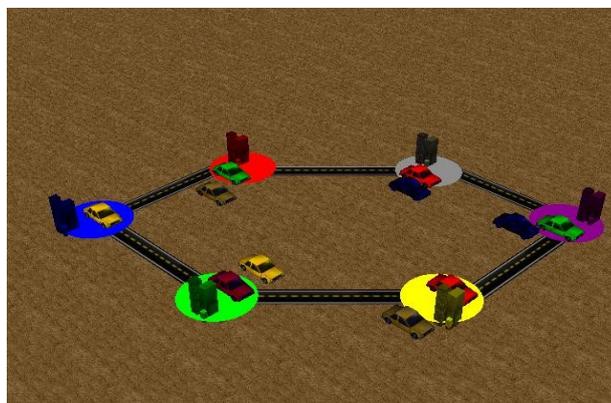


Figure 7: Whilst a deadlock example similar to this could be created in Alice, many of the difficulties similar to the sorting network concept occur

(<http://csunplugged.org/routing-and-deadlock>) could be implemented. This was done in the early stages of design and a decision was made to move on from this concept due to the similarity in problems to the sorting network concept. As with the sorting network, it was found rather tedious to place the objects in the world, but the ability of Alice to change the colour of any model makes it rather easy to use similar objects but alter them so, in the case shown in Figure 7, cars of a specific colour must reach buildings of the same colour – similar to the concept of a router needing information destined for it.

Another concept briefly investigated was using Alice to show simulations of ideas such as information hiding (<http://csunplugged.org/information-hiding>). While simulations like these are often possible within Alice, this idea was dropped due to the fact that many of the implementation ideas would either cross over with other activities, or problems with difficulty in defining a suitable implementation.

4.3. Features and Issues Within Alice

During this case study a number of potential benefits and problems with Alice were found. Many of these have been mentioned in the concepts themselves, such as the visualisation objects and variable referencing. This section discusses problems and benefits that have not been previously discussed. A number of these issues, as well as some not mentioned here, have also been brought up by Dick Baldwin, a professor at Austin Community College, on his website (<http://www.dickbaldwin.com/alice/Alice0920.htm>).

4.3.1. Alice is not Object Oriented

Although Alice is meant to be a first exposure into object oriented (OO) languages, it itself is not object oriented, but object based. Concepts missing from Alice such as polymorphism and inheritance are vital parts of OO languages, and although Alice does have encapsulation, it does not enforce it. Although not directly OO-related, Alice also does not allow casting, which can pose other issues when trying to create worlds. This may not be a negative feature, though, and [6] discusses how this method works well as an introduction into OO languages, and also reflects on many other advantages of Alice.

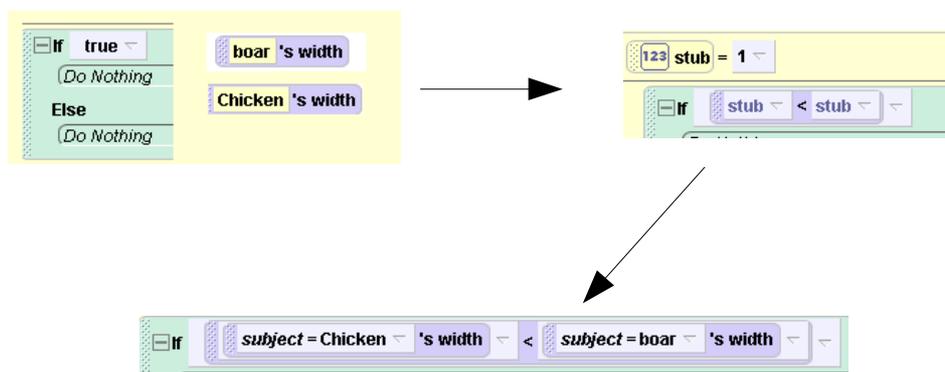


Figure 8: What should be the simple addition of variables into an comparison statement can become a multiple step task rather easily

sky with the “when I click anything” event mentioned earlier.

4.3.4. *Basic Features Missing*

An interesting issue in Alice is the inability to delete various components once they are created. If a user creates a method, then that method is there to stay, with only the option to rename the method available. This problem only occurs in a few specific places, but can make code feel messy.

4.3.5. *Array and Visualisation Object Bugs*

Both the standard arrays and the array and list visualisations in Alice have a number of bugs and interesting features that can cause additional frustration while creating programs. The most major of these would likely be the fact that array sizes can increase, but not decrease, as similarly to the last point there is no delete feature. Basic array sizes cannot be changed at runtime, but when an array is created through the drag and drop interface you start with an initial array with one item inside it. From here, you can create additional array slots, but if you later find that an array is too large you cannot remove these – instead the array must be deleted and recreated, or the remaining slots left empty with the appropriate programming to ensure these unused slots are never referenced.

List visualisations also have an awkward “remove” feature. When an object is removed from the list, it often disappears, and at times will reappear later if that array slot is reused. This hidden object never actually leaves the virtual world, which is not necessarily a bad thing, but this default disappearing behaviour can be quite awkward.

4.3.6. *Processor Usage*

Alice has an interesting problem with using large amounts of processor capability, to the extent where it can run a processor at 100% indefinitely. While significant investigation as to why Alice does this was not performed, this problem often caused the need for Alice to need to be closed and reopened to continue usage or prevent processor overheating. It was noted that this problem would occur more often with other programs open, and would also often occur while Alice was sitting idle in the background.

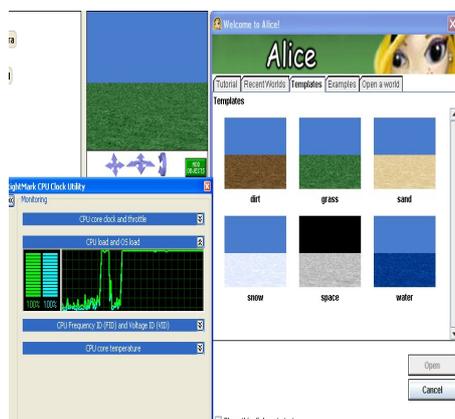


Figure 10: Alice upon being freshly opened

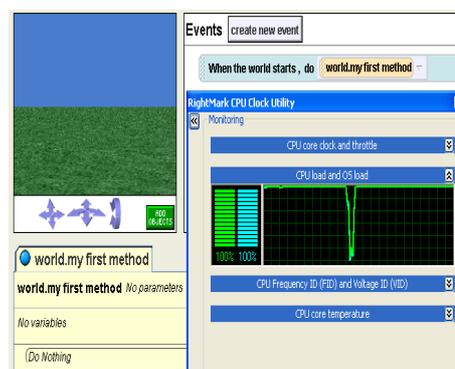


Figure 11: After opening a new, blank world Alice continued to run at peak processor usage

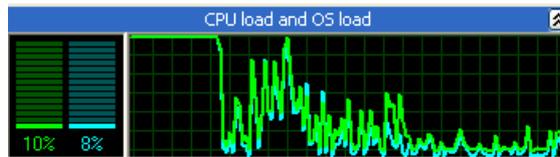


Figure 12: Upon closing Alice, processor usage returned to normal

Regrettably, even upon starting a fresh Alice application this can happen, as seen in Figure 10. In this case, Alice was opened, and left to run while RightMark (<http://cpu.rightmark.org/>), a CPU monitoring program, kept track of CPU usage, which can be seen as a line graph in Figures 10, 11 and 12. After a while, the initial Alice screen was closed, opening a new, empty Alice world, at which time processor usage dropped temporarily before spiking again, as seen in Figure 11. Finally, upon closing Alice, processor usage dropped back to normal levels (Figure 12). While in this case Alice began causing this bug from startup, sometimes the program can efficiently be used for long periods of time before such problems occur.

4.3.7. Saving and Loading

As mentioned within the concept investigation, Alice represents each object as an individual object of its own unique type. Regrettably, it also does this with each model, causing Alice to save a large amount of unneeded data, and also causing save times to become significantly large when there is a high number of complex models to save.

Simple objects, such as a single ball or cube, can occur in large numbers in an Alice world without significant issue. More complex objects such as a penguin, which has a model with a number of complex parts and many unique pre-defined methods, can quickly overwhelm the system, causing save times to increase to several minutes, and on occasion causing Alice to crash while saving, corrupting the save file in the process. In the case of the penguin model, any number over 100 could cause Alice to crash unexpectedly, whilst at this point 100 ball objects would still only need a save time of a few seconds.

4.3.8. Numbers and “.0”

Numbers in Alice are represented in a decimal format, and as shown in Figure 6 by default a “.0” is added to any number printed, whether this be using the 3D Text object, a printed output, or the “say” or “think” method of an object. After searching both Alice and the community forums for a method of removing this additional decimal digit there was no simple option to remove it, even adding extra statements to get a “integer only” option to appear, such as in the random number functions, failed to remove this addition. While this issue is minor, in the case it is seen in Figure 6, it can distract from the nature of number representations such as in the binary calculator

4.3.9. Alice and Storytelling

Alice provides a number of additional methods and classes which are helpful for creating story-like animations. Methods such as “say” or “walk”

are amongst many functions that, while not exactly appropriate to creating CS concepts, give the ability for a user to let their imagination run wild. There is another version of Alice known as *Storytelling Alice* (<http://www.alice.org/kelleher/storytelling/>) which enhances some of these features.

4.3.10. Video Export

Alice provides the feature to export the animation of a virtual world as a video in the .mov format. To save a video, Alice records the animation as it happens, meaning any interactive components will be exported in the video as they are entered during the run sequence. This also means that with one single interactive world several different videos could be saved, showing how an algorithm may change with different inputs.

4.4. Discussion

Although a number of issues with Alice have been found and discussed, Alice does have the potential to be a strong system for teaching computer science concepts such as those discussed here. Many of the current bugs can be avoided through knowledge of where they lie in the system, and once the location and knowledge of features like array visualisations, many concepts can relatively easily be implemented.

The success of being able to implement sorting algorithms such as quicksort with minimal additional code provides a strength that other visual programming languages would have difficulty to do as efficiently. Tools like the array visualisation are a step in the right direction for simplifying the additional complexity added by visualising a concept over just implementing it in a non-visual way.

Alice performed all three concepts rather comfortably after methods of implementation were found. The sorting network, while not automated, showed that Alice both provides methods for making a number of basic programming constructs available and also allows for objects in the virtual world to be relatively easily placed and moved, barring a few issues with the world not being flat. The sorting algorithms showed that Alice provides the means to animate arrays without increasing the amount of code above that of non-visual array implementation, and the implementation of a binary number calculator proved that the event binding in Alice is both simple and strong, with only a few small bugs letting it down. The drag and drop in Alice also means it is easily accessible, with little knowledge of the system needed to create some basic programs, and little time required to learn the system as a whole.

At the time of this report Alice 2.2 was still being worked on, with the hopes that a stable final version would be released. Although it has been stated that there will be no major functionality changes, if a number of the major bugs above are fixed Alice will become both a powerful, but simple to use language. Even in the current build, Alice definitely provides the capability to produce programs that show concepts visually without substantial extra coding, and could provide methods for implementing a large proportion of

content needed in upcoming curricula changes without requiring students to learn a more complicated language like Java until later years.

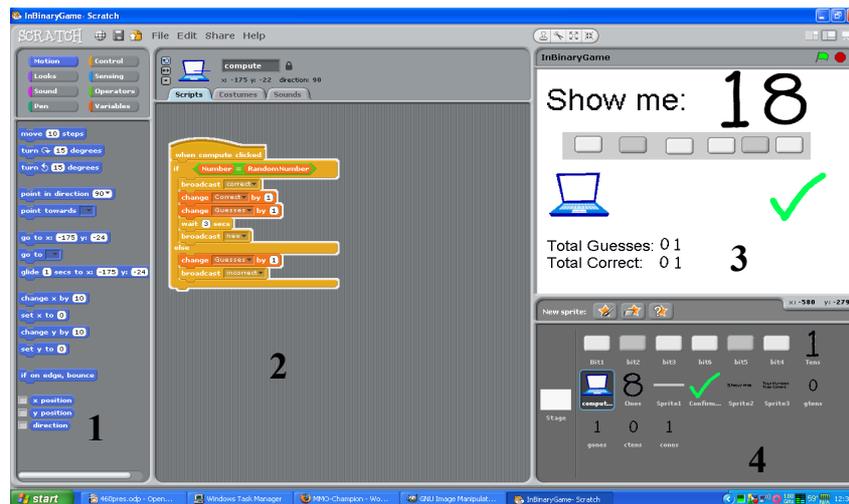


Figure 13: The Scratch interface, comprised of available code blocks (1), the current open script (2), the virtual world (3) and the object list (4)

5. Scratch

Scratch is a simple yet versatile 2D virtual programming language. Currently, it is used around the world to create games, teach concepts from various subjects, and teach programming fundamentals. Scratch also provides the capability to share projects online, not only allowing for project code to be downloaded, but also giving the ability to show projects on the Scratch website. This language is aimed for users of age 8 and above, although it is also used by many secondary institutions.

The Scratch interface, seen in Figure 13 with various labelled parts, is relatively similar to a typical Integrated Development Environment (IDE). It has a code editor (part 2), a list of all objects in the world (known as “sprites”, shown in part 4), a list of code blocks which can be used (part 1), and the virtual world itself (part 3). Code blocks come in eight different categories, each represented by a different colour.

Scratch, much like Alice, uses a drag and drop style approach. However, Scratch makes it significantly easier to tell which code blocks fit where, and what type of code block (for example, a control block or a motion block) it is by the use of shape, in the form of jigsaw-like pieces and colour respectively.

Also like Alice, a significant amount of research has been done into the effectiveness of using Scratch. One key paper is [10], which discusses how Scratch motivates students and familiarises them with programming ideas without overwhelming them. Another notable paper is [15], which reinforces a number of these ideas.

For this study, Scratch 1.4 was used, although some initial work was done in a previous version. Some newer features such as the camera block and LEGO WeDo™ support were not investigated.

5.1. The Concepts in Scratch

Scratch, whilst being a much simpler language to use than Alice, proved to have less features and constructs available. This made it significantly more difficult to produce the desired concepts easily. Scratch did, however, have a specific type of implementation where it was found that not only were the concepts produced, but that they were produced without the need for significant additional complexity while easily fitting within the Scratch world – the binary number games.

5.1.1. *Sorting Network*

Because Scratch is a 2D world, and also gives the ability to create sprite images within Scratch (or import images as sprites) it was much easier to create and place the model of the sorting network. There are multiple methods for doing this – whether the network is split into parts or build as one solid sprite, or as a background for the characters to sit on.

In Figure 14, each circle, square and line is an individual sprite, as well as a sprite for each character sitting on a node. Using this method, there is an issue of how Scratch layers sprites. Since the sprite most recently interacted with manually is considered the top layer sprite, it can cause issues if a sprite needs to be moved. If any of the non-character nodes are moved, then the program run, and character will appear under the moved nodes if they move to them. This issue is easily resolved by simply moving the character manually or using the “show” function. If the sorting network is drawn as the background these issues would not occur, but benefits of being able to move directly to another sprite would be lost.

As with Alice, creating a network that uses pre-defined numbers is rather easy. Scratch recently added the ability for a user to input String type variables into a program, and could possibly use this to make an automated version, but the lack of an object array construct increases the difficulty in producing this implementation.

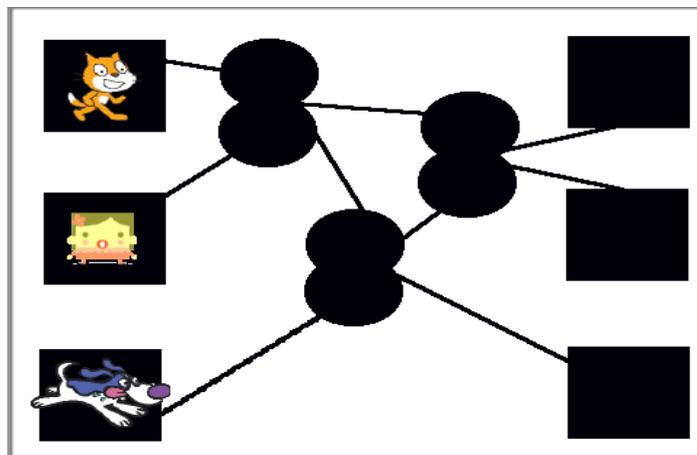


Figure 14: A basic 3-node sorting network in Scratch

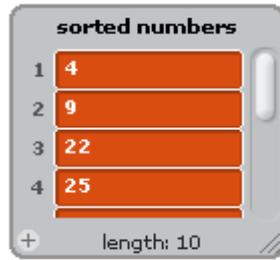


Figure 15: Scratch allows all variables and arrays to be visualised in a textual manner

5.1.2. *Sorting Algorithm*

It was significantly more difficult to implement sorting algorithms in Scratch, and more-so difficult to visualise these due to the lack of an array that could contain sprites.

A simple sorting algorithm that shows a given set of numbers sorting is quite possible in Scratch, however, as Scratch provides a textual array similar to that shown in Figure 15. Sorts that use recursion, such as quicksort, cannot be done in Scratch in the conventional ways due to the lack of parameters and recursion. Although it may be possible to implement such sorting algorithms using the “broadcast” and “when I receive” blocks, this was not explored due to the additional complexity it would add to any students recreating the same implementations.

5.1.3. *Binary Number Operations*

As previously mentioned, Scratch performed very well in creating game-like binary conversion and parity simulations. A binary calculator itself was easy to implement, and examples of such concepts working in Scratch were found on the Scratch website when searching for “binary” (<http://scratch.mit.edu/tags/view/binary/views/>), so this lead into trying some other ideas of binary conversion games that could be created, yet still show this concept in action. A binary calculator was still created to explore implementation ideas (Figure 16, left), with a simple implementation of bits similar to those done in Alice.

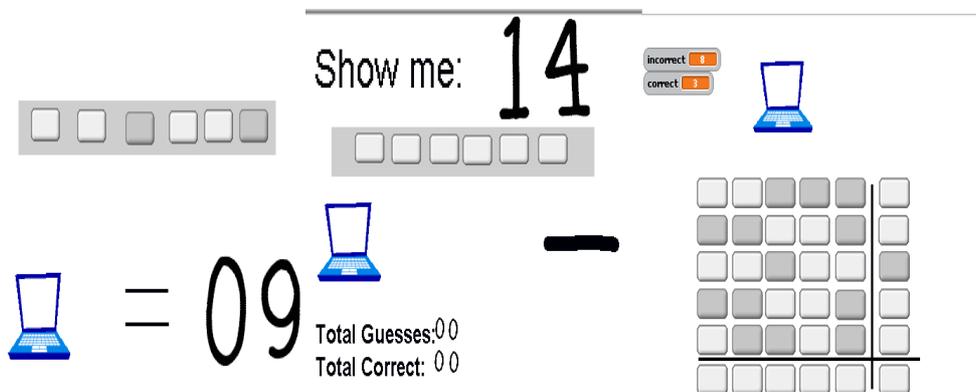


Figure 16: Three of the worlds implemented in Scratch, representing the binary converter, binary guessing game and a parity guessing game

A binary guessing game was created (Figure 16, middle), where the student was given a number and asked for the binary representation, whilst a scoreboard of correct and incorrect answers kept track of correct and incorrect guesses. The number represented was altered each time a correct answer was given, and correctness computed when the computer icon was placed. A number of additional scripts needed to be coded to create this game, including using two number sprites to represent one double digit number, a notification for answer correctness (shown as a black line), and variables and displays for the number of correct and incorrect guesses. In each case the scripts were rather simple, consisting of no more than a few blocks of code.

The parity game, shown in Figure 16 (right) was also rather simple to produce. This game sets all non-parity bits in the grid to random states, and calculates what the required state of the parity bit needs to be. From here a user is asked to set these states, with a number of correct and incorrect bits shown at each computation. Because of Scratch's lack of way to iterate over a number of sprites easily, every bit contained duplicate code that could have easily been removed if a sprite array was available. Scratch also lacks multi-dimensional arrays, which would have been helpful in creating this concept.

5.2. Features and Issues Within Scratch

5.2.1. *World Size*

One problem with Scratch encountered was the limitation of the size of the virtual worlds. Currently, the virtual world is set to a 480 by 360 pixel area, which makes it difficult to show anything substantial within the virtual world, such as a large number of sprites being sorted had sprite sorting been successful. Scratch does allow for scrolling and other similar techniques to make a world seem bigger than it is, however.

5.2.2. *Language Limitations*

Scratch was also limited in the number of complex constructs it had available. While it has basic constructs such as loops, if statements, and some complex event binding facilities, it lacks other common types.

Mentioned previously, arrays became one area that this crept up. In each concept there was a point where iteration of sprites through an array would have been a helpful feature. Using awkward sprite naming conventions, such as adding a number to each sprite name, and iterating through sprites using that reference was one possible way or working around this, but this produces a lot of undesired complexity.

Parameters are another feature that Scratch is lacking, and there are a limited number of ways to tell a script in Scratch to run given a certain operation occurring, but there is no direct way to tell a script to run while using a particular variable. The use of broadcast and receiver blocks allow for, to some extent, parameters to be mimicked if global variables are used, but parameters themselves are not directly implemented. The use of event

bindings and text input can be used in similar ways.

Variables themselves are also very limited in Scratch, with only one basic variable implemented – a String, created by the “Make a Variable” button. Whilst Scratch allows this String to also be used as an Integer type for calculations, it does not strictly provide an actual Integer type. These variables are also provided with basic blocks such as “change by”, where the variable is increased or decreased by a given integer, as well as blocks such as “set” having defaults that are numerical. Having the 2 variable types available split into two separate types – a String and an Integer – would be much less confusing.

5.2.3. *Variable Initialisation*

An interesting feature, variables in Scratch, unlike those in Alice, do not reset to their initial state when the “stop all scripts” button is pressed, or when the program ends naturally. Instead, they remain in their final state. This feature has both advantages and disadvantages, as it helps teach the concept of setting variables in code so they are set to known values when a program starts, but also adds complexity as the user will have to set a number of variables they may not consider, namely those variables all sprites has by default, such as location, size and orientation.

5.2.4. *Redundant Coding*

Currently, Scratch has no way to create more than simple arrays containing strings of text or numbers. This can be a problem when trying to make several sprites do the same, or very similar, things that could otherwise be done via array iteration. This makes it rather easy to produce code duplication, as in some cases it becomes much simpler to drag and drop large sections of code rather than producing a working set or broadcasting and receiving scripts.

5.2.5. *World Sharing*

Rather than give some sort of complex export feature, Scratch chooses two simpler options. The first is the ability to share projects online via the Scratch website. The second is a presentation mode within the Scratch interface, which gives a full-screen view of the current world to present on a projector or larger screen.

5.3. Discussion

While Scratch could provide the needed tools to illustrate a CS concept such as those discussed in action, it is limited in both the fact that the amount displayed within the virtual world is small and that many key programming concepts, such as object arrays, parameters and other variable types, are missing or only somewhat provided.

Although it was able to fulfil each concept to some extent, the time spent on finding a suitable implementation of the sorting algorithm example did not lead to an outcome that was highly desired. Although Scratch can sort arrays

using some algorithms, the only simple way found to do this is via the textual form of an array, as adding additional visual constraints removed the possibility of doing this concept within an array.

Scratch shows promise at making game-like simulations. The creation of a “show this number in binary” game was not a planned outcome, but because of how simple Scratch made both the creation and handling of the scripts used it became a natural progression from the basic binary calculator.

Overall, Scratch was found to be a simple, easy to learn language. Although, from this study, it seems that Scratch could not provide ways to implement a reasonable proportion of concepts in the upcoming curricula, as an introductory language it could provide a good starting point for those who have very little programming experience.

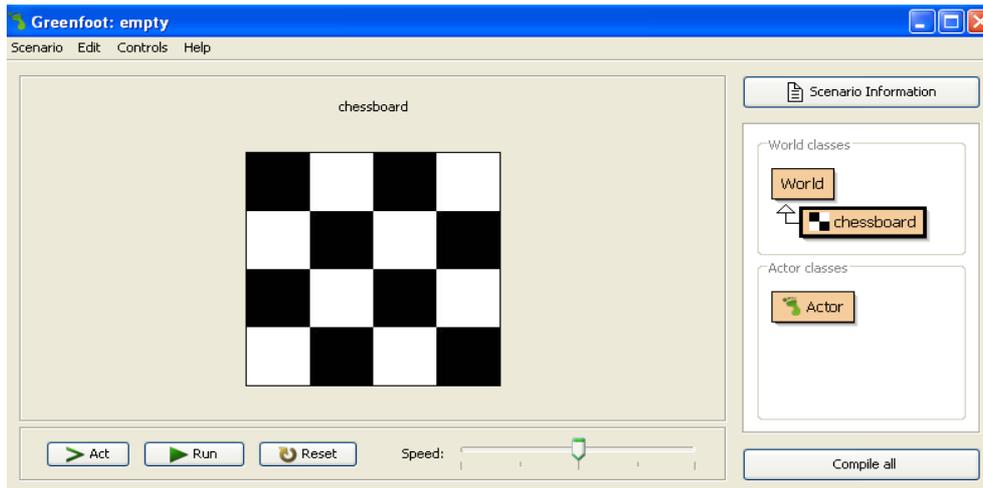


Figure 17: The Greenfoot interface is rather minimal compared to languages like Alice and Scratch

6. Greenfoot

A combination of a framework for creating 2D virtual worlds and an IDE, Greenfoot is aimed to create a way for novice programmers to easily create exercises that require a visual element.

Greenfoot uses Java as a language, and is built similarly to BlueJ (<http://www.bluej.org/>), but with a virtual world and some default classes added to the interface. This means, unlike the other two languages, aspects of programming like syntax are yet again important, and the full power (and complexity) of a language like Java is available to use. Greenfoot is aimed at ages 13 and above, although because it uses a language like Java it can quite easily be used in university-level courses.

Even with these higher requirements, previous research into Greenfoot has shown that attitudes improved towards programming over students that had little to no programming experience [2]. Greenfoot is designed to enhance support for both teachers and their students, making scenarios and exercises more manageable than traditional languages [8].

The interface, shown in Figure 17, simply contains the world or worlds, basic options for running them, and a tree of all classes within the program. Compared to Alice and Scratch, this is rather minimal. Code is opened in a rather plain text editor, that uses simple colour coding, in a new window.

The version used in this study was 1.5.6.

6.1. The Concepts in Greenfoot

Although there were time constraints, all concepts were implemented to some extent in Greenfoot. Although Greenfoot uses Java as a language, no changes to concept implementation were changed to take direct advantage of this. Using the findings of the concepts tried, some discussion of those not completed has been added to the relevant sections.

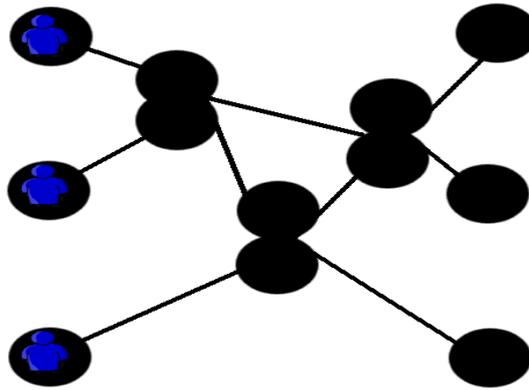


Figure 18: An implementation of a sorting network in Greenfoot looks very similar to those made within Scratch

6.1.1. *Sorting Network*

Because of how Greenfoot deals with objects, it was much simpler to create an image of the sorting network and use that as a reference, as seen in Figure 18. An attempt was made to create “node” and “path” objects to create the world, but it was found more unwieldy than creating the same network mapping in a language like Scratch, where coordinates for each node could easily be read off the world from the current mouse position, whereas Greenfoot did not seem to provide such an option without additional programming, meaning trial and error through using the “setLocation” method needed to be used.

A character class was created, with a blue person image used to represent it in the virtual world (see Figure 18). This class represented the item to be sorted, and given a number to be sorted by. This class contained only the variable, a constructor to initially set this variable, and a way to compare two different characters. Although the implementation of this was not finished, the use of the “setLocation” method would make it easy for characters to move through the network.

6.1.2. *Sorting Algorithms*

This concept, whilst requiring a higher level of coding skill than Alice or Scratch to make, was rather simple to implement. In this case, there were two classes, a subclass of the world to containing the sorting algorithm itself, and a single actor subclass to represent a sortable item. The sortable

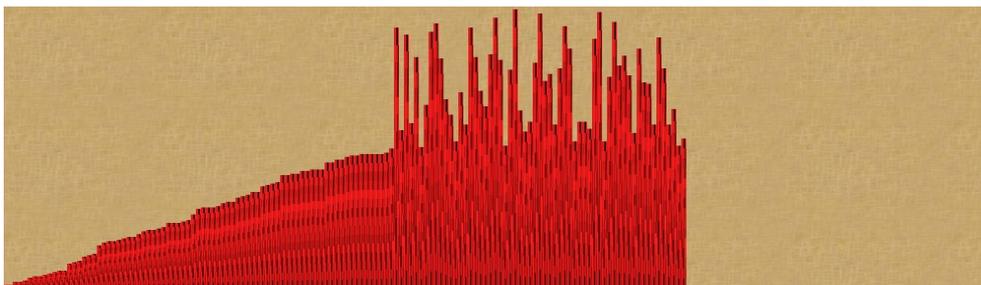


Figure 19: A partially completed selection sort in Greenfoot. Unlike Alice and Scratch, it is rather easy to create large number sorts like this without losing view of part of the array

item had only a constructor that was passed a height parameter, and a single method to compare the height with another sortable item, whilst the subclass for the world additionally contained methods to create randomised sortable items upon each use of the program.

Because of Greenfoot's ability to easily scale an image, through the `getImage().scale(int, int)` method, this allowed large sorting networks to be easily created. In Figure 19, a selection sort in progress of an array of over 100 varying height items can be seen, although by changing both the sizes of the objects being sorted and increasing the size of the world Greenfoot can quite easily show larger numbers of items being sorted. This could be generalised to sorting objects such as different animals.

6.1.3. *Binary Number Operations*

The attempt to implement a binary calculator was rather successful, as much like Alice and Scratch, Greenfoot provides a simplified way to implement mouse-bound events through a “MouseInfo” class. Through this class, a click event can easily be added to the “act” method of the world, which flips the correct bit, altering any values required. As with the sorting algorithm example, only two new classes were created, with an additional class called “Counter” taken from the Greenfoot support classes, found on the Greenfoot website.

Figure 20 shows an implementation of such a binary calculator. In this case “1” bits are represented as a green circle, and “0” bits represented as a red circle. When a circle is clicked, the colour of the bit is changed, and the counter, in this case with a string called “output”, updates the current total.

A parity game would be a rather simple expansion from this binary calculator. Unlike Alice and Scratch, Greenfoot has access to multi-dimensional arrays, reducing the need for extra, complex coding that these two other languages had.

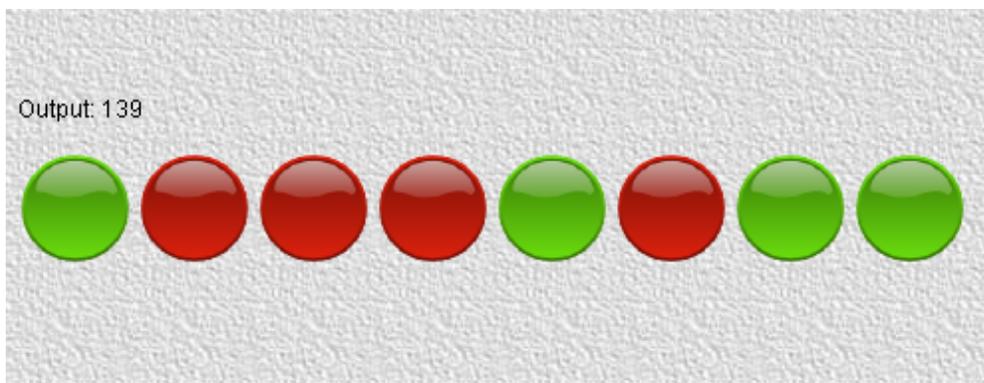


Figure 20: A simple binary calculator in Greenfoot is rather simple to implement due to Greenfoot's MouseInfo class

6.2. Features and Issues Within Greenfoot

6.2.1. *Higher Knowledge Requirement*

Although Greenfoot is aimed at a rather young audience, the requirement of Java knowledge, including aspects such as syntax, are often not taught in other languages geared towards this age level. Whilst it would be rather easy to give a student pre-defined code for aspects such as randomisation and other parts of each program not related to the concept, and ask them to create the methods for the concept-related part (such as the quicksort or selection sort algorithm), the user would still need an understanding of how to implement such parts in Java, whether they be recursion, simple loops, or comparisons, whilst in comparison it would be much easier to ask a student to implement all functionality without additional scaffolding, as additional features such as randomisation would not require students to search outside of the interface to find them.

6.2.2. *Greenfoot is less self-contained*

Compared to the other languages studied here, Greenfoot chooses to minimise the available information. The interface itself is rather basic, providing only the simple world overview and class layout, and basic references to the Java documentation of Greenfoot's classes. This means a student will likely need to leave the Greenfoot environment to find the needed functions, files and other important information, often requiring access to sources such as Java API or support classes found on the Greenfoot website to complete tasks.

6.2.3. *Export Features*

Greenfoot allows many types of export, including uploading projects to an official sharing website, producing a .jar file of the world, and creating a webpage containing an applet of the project. When a .jar file is exported, the program works just as it did in Greenfoot, but without the “act” button or listings of classes and code.

6.2.4. *Adjustable World Size*

Unlike Scratch and Alice, Greenfoot allows the world size to be defined by the user. A grid of cells is used, with each cell being of a user-defined number of pixels, as well as the width and height representing the number of cells in this grid. As locations are represented by cell location, it can cause issues with finding positions within a world – for instance in one world a location of “5,5” will seemingly represent a different world's location of the same values if that world has a different cell size. One simple way to get around this issue is to deal with every world in pixels by setting the cell size to “1”. Thus an 800x600 world will represent 800x600 pixels, each with their own individual location.

6.2.5. *Support Classes*

As mentioned in the binary number concept, the Greenfoot website provides a number of helpful support classes to prevent the need for programming aspects such as animation and movement. Classes like these could also be created as required for activities to reduce difficulty or the need for unrelated programming concepts to be implemented, unlike Alice and Scratch which provide little to no ability to import simple scripts or methods from other projects.

6.3. Discussion

Greenfoot is a very promising language. Although it requires reasonable knowledge of Java, also causing it to require the knowledge of syntax, the additional power given by this increased complexity easily allows it to create all of the concepts mentioned here, and likely most, if not all of those concepts outlined in upcoming curricula.

As an introductory language, this may make it less suitable to teach concepts without requiring a reasonable knowledge of Java. Students could have a level of scaffolding in place with each exercise to help reduce this required knowledge, as well as creating exercises that do not require additional Java classes that may require students to venture into the Java API, instead focusing on using classes of the Greenfoot API as well as pre-written classes to avoid this. Even with these, the need for syntax knowledge would still be a significant learning step over other languages. Because of this, it is recommended that Greenfoot would either require a more careful approach as a first language, or be more appropriate as a language following an introduction to programming from a language such as Alice.

7. Future Work

In addition to the concepts and languages explored here, there are a wide range of other possibilities that were not fully investigated. During the course of the study, some of these were looked at to see if they would provide an improvement over existing ideas or options, while others were not highly considered and need significant further investigation. As a number of other languages were briefly explored, this section focuses on those rather than other concepts that may be worth investigating.

7.1. Other Promising Languages

In addition to Scratch, Alice and Greenfoot, a number of other languages seemed to show themselves as having potential to fulfil the tasks required by this study, but themselves were not a part of it for various reasons. Amongst these were the Alice 3 beta, Kodu, and LOGO. Using a language such as Second Life is also discussed.

7.1.1. *Alice 3*

While a full case study of the Alice 3 beta was not completed, some initial investigation was done to determine if, in the current state, if there was a significant improvement over the bugs and missing features from Alice 2.2. Beta build 60 was studied.

Alice 3 takes a different approach to Alice 2.2. Most noticeably, the interface lacks the event binding section, replacing it with a listener style more similar to that of Java. Other ideas take a more Java-oriented approach, too. To some extent, Alice 3 is a completely different language to previous versions, as much of the programming style has changed, although many of the drag and drop aspects, as well as the style of virtual world, remain the same.

Unfortunately, Alice 3 currently does not include the “Visualisation” objects that made many of the concepts significantly easier to implement in Alice 2.2. A number of other features have also been removed, such as image and sound importing, video export, and many, many more. There were also a number of bugs found while testing the beta that did not allow some basic but important functions to be used. Changing a variable of a character, such as their height, would change the height of the character visibly inside the virtual world, but would return the number related to the size unchanged, meaning a sort algorithm over people with manipulated height variables would not visually work correctly.

Alice 3 does have one significant new feature, though, through a Netbeans plugin. Using this Alice code can be uploaded into Netbeans and be edited in a Java environment before being returned in an Alice environment, allowing some functionality that is otherwise difficult to attain in Alice to be produced.

Overall, this beta still needs some significant work to bring it to the level of Alice, and this is why a full study of the system was not carried out. It

shows promise in bridging the current gap between Alice and Java, as well as fixing some of the issues that are unlikely to be fixed in Alice 2.2, but the loss of features such as visualisations is disappointing given how relevant they were found in this work.

7.1.2. *Kodu*

Kodu, from Microsoft, is a visual programming language specifically geared towards creating games, aimed at young children. The language used is entirely “icon-based”, with complex ideas such as collision detection simplified into a single icon. While this language may be very appropriate for teaching concepts that can be created in a game-like manner, it is also currently only available on the Xbox platform, with a PC version currently unavailable. Because of this Kodu was not chosen for this study, although a study of the language would prove interesting.

7.1.3. *LOGO*

LOGO is another programming language that could have been investigated, and it has a wide number of implementations that incorporate some aspect of visual programming. But due to the sheer number of implementations available, it would be impossible to choose one effectively without appropriate research into the others, thus LOGO would probably require a significant study of its own to determine which implementations were worthwhile using.

7.1.4. *Second Lifes LSL*

Initially, this project had intended to look at the Second Life language, LSL, and how it compared to Alice as a virtual world. Due to various ideas while exploring Alice, as well as the noted changes to curricula, Second Life was dropped in favour of languages similar to Alice in characteristics. Second Life also has several other issues, such as the cost of purchasing land, ethical issues tied to children interacting with adults in a multi-person virtual world, and other factors that were less appropriate to school-oriented teaching.

8. Conclusions

Although each of these languages has been shown to have advantages and disadvantages, they can all provide potential in creating a number of CS concepts, both those shown here and those which would need similar tools to design.

Alice provides a well rounded tool, and although it has a number of bugs these can often be avoided. In the current state, Alice can be difficult to use, but with knowledge that major bugs are being worked on, and the fact that with knowledge of the system, projects can be made that avoid many of these issues. Because the language does not require any significant programming knowledge to produce working code it would make a good introductory language to learn both the concepts of programming, and the concepts of CS.

Similarly, Scratch would be suitable as an introductory tool, but unlike Alice would likely require a more powerful language to be learnt at an earlier stage. The lack of some key programming constructs is a disappointment, as with some of these more advanced features, such as object arrays and direct parameter passing, it would compare with Alice in complexity, but provide a much more stable environment.

Greenfoot provides itself with the most powerful set of tools by being an extension to the Java language, allowing some complex programming aspects such as event binding to be done with a minimal amount of code compared to normal Java implementations. But this is also a disadvantage, and given this extra complexity it would take extra work to both learn the additions of the Greenfoot language, as well as any Java needed to implement the given concepts. Greenfoot would work well as a language for those with some experience in programming, but needing an interface that both allows for the full complexity of Java, but enabling the ability to minimise this through support classes.

Of course, each of these languages has something they excel at over the other two. Alice performs best when it comes to animation, with a superior 3D virtual world and wide range of character animations predefined, but this is of course undermined by the number of bugs and instabilities within the software. Scratch has a simple yet clear implementation, allowing more complex games and animations to be easily created, but lacks the programming constructs to produce complex algorithms. Scratch was also the language that seemed the easiest to just install and use. Greenfoot has both the power and complexity of Java, which is both a huge benefit and a significant disadvantage, and would need careful implementation of activities to prevent overwhelming new students.

To put a rough comparison on where each language sits relative to the others, their recommended age ranges fall rather close to where each language is in difficulty. Scratch is clearly the language that would be most suitable to new and young learners, although Alice is not far behind and the problems with the increased difficulty lie more in knowing how to avoid and react to bugs and other software instabilities than an actual increase in

language difficulty. Greenfoot would easily be the the most difficult of these languages, and although the recommended age is put around 13 it would probably do well as a language at a higher age bracket.

This report has looked at how Alice, Scratch and Greenfoot perform when put to the test of visualising specific computer science concepts. It has shown how each of these languages can, to some extent, perform a number of varying implementations of these aspects, and what limitations on the languages, given their language's constructs and bugs in the systems. Each of these languages provides an environment in which many of the concepts outlined in various upcoming curricula are implementable in a rather simple fashion. While each system discussed has a limitation on either the amount of these concepts, or ease of implementing them, each of them would prove rather fitting as early programming languages in these modified curricula, and the possibility of using a pair of these languages, one for new students and one for advancing students, would allow most, if not all, concepts to be covered over the several courses outlined.

9. References

1. Al-Bow, M., Austin, D., Edgington, J., Fajardo, R., Fishburn, J., Lara, C., Leutenegger, S., and Meyer, S. 2008. Using Greenfoot and games to teach rising 9th and 10th grade novice programmers. In *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games* (Los Angeles, California, August 09 - 10, 2008). Sandbox '08. ACM, New York, NY, 55-59. DOI=<http://doi.acm.org/10.1145/1401843.1401853>
2. Bell, T., Andreae, P., Lambert, L. *Computer Science in New Zealand High Schools*. ACE January 2010
3. Brown, P. H. 2008. Some field experience with Alice. *J. Comput. Small Coll.* 24, 2 (Dec. 2008), 213-219.
4. Computer Science Teachers Association, Results of the CSTA National Secondary Computer Science Survey (2009) - <http://www.csta.acm.org/Research/sub/CSTAResearch.html>
5. Cooper, S., Dann, W., and Pausch, R. 2003. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Nevada, USA, February 19 - 23, 2003). SIGCSE '03. ACM, New York, NY, 191-195. DOI=<http://doi.acm.org/10.1145/611892.611966>
6. Dann, W., Cooper, S., and Pausch, R. 2001. Using visualization to teach novices recursion. In *Proceedings of the 6th Annual Conference on innovation and Technology in Computer Science Education* (Canterbury, United Kingdom). ITiCSE '01. ACM, New York, NY, 109-112.
7. Dann, W., Cooper, S., and Pausch, R. 2008 Learning to Program with Alice (2e). Prentice Hall Press.
8. Henriksen, P. & Kölling, M. 2004. Greenfoot: combining object visualisation with interaction., in John M. Vlissides & Douglas C. Schmidt, ed., *OOPSLA Companion*, ACM, pp. 73-82
9. Lorenzen, T. and Sattar, A. 2008. Objects first using Alice to introduce object constructs in CS1. *SIGCSE Bull.* 40, 2 (Jun. 2008), 62-64. DOI=<http://doi.acm.org/10.1145/1383602.1383636>
10. Malan, D. J. and Leitner, H. H. 2007. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA, March 07 - 11, 2007). ACM, New York, NY, 223-227.
11. Moskal, B., Lurie, D., and Cooper, S. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA, March 03 - 07, 2004). SIGCSE '04. ACM, New York, NY, 75-79. DOI=<http://doi.acm.org/10.1145/971300.971328>
12. Mullins, P. M. and Conlon, M. 2008. Engaging students in programming fundamentals using Alice 2.0. In *Proceedings of the*

9th ACM SIGITE Conference on information Technology Education (Cincinnati, OH, USA, October 16 - 18, 2008). ACM, New York, NY, 81-88.

13. Mullins, P., Whitfield, D., and Conlon, M. 2009. Using Alice 2.0 as a first language. *J. Comput. Small Coll.* 24, 3 (Jan. 2009), 136-143.
14. Tucker, A. (editor), Deek, F., Jones, J., McCowan, D., Stephenson, C., and Verno, A. A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force Curriculum Committee. Association for Computing Machinery (ACM), New York, New York, October, 2003 (Second Ed., 2006)
15. Wolz, U., Leitner, H. H., Malan, D. J., and Maloney, J. 2009. Starting with scratch in CS 1. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA, March 04 - 07, 2009). SIGCSE '09. ACM, New York, NY, 2-3. DOI= <http://doi.acm.org/10.1145/1508865.1508869>

10. Appendices

The following programs were created and used as a part of this study: Many of these are available on the CSUnplugged sharing group (<http://groups.google.com/group/cs-unplugged-sharing/files>), as well as on the Computer Science Education Research Group (CS_ED) Learn page (<http://learn.canterbury.ac.nz/course/view.php?id=961>)

The file names below represent examples of the concepts explored.

Alice:

Sorting Networks:

- alice-ThreeSortExample.a2w

Sorting Algorithms:

- alice-selectsort.a2w
- alice-Qsort.a2w
- alice-UnimplementedSortWorld.a2w

Binary Calculator:

- alice-Binarycalc.a2w

Scratch:

Sorting Networks:

- Scratch-NetworkBase.sb

Sorting Algorithms:

- Scratch-sort.sb

Binary Number Games:

- Scratch-BinCalcFinal.sb
- Scratch-InBinaryGame.sb
- Scratch-paritygame.sb

Greenfoot:

Sorting Networks:

- Greenfoot-SortNetwork.rar

Sorting Algorithms:

- Greenfoot-sortalgorithm.rar

Binary Calculator:

- Greenfoot-BinaryCalc.rar