Abstract
This paper examines implementing common computer science problems in Alice and Scratch.

INTRODUCTION
Initial learning environments (ILEs) such as Alice (www.alice.org), Scratch (scratch.mit.edu) and Greenfoot (www.greenfoot.org) make it possible to access the power of programming in a visually appealing environment without a comprehensive understanding of syntax and object-oriented programming concepts. At the same time, new models of Computer Science curricula, including CSTA’s K-12 model curriculum [CSTA06], promote teaching Computer Science concepts early in a student’s Computer Science education to help them appreciate the depth and breadth of the field. Initiatives such as CSUnplugged (csunplugged.org), CS inside (csi.dcs.gla.ac.uk) and CS4FN (cs4fn.org) expose students to a broader view of CS in non-programming environments.

This paper investigates implementing non-programming activities based on computer science concepts in two ILEs (Scratch and Alice).

RELATED WORK
Recursion [Dan01], binary search [Mul08], and some mathematical concepts such as set operations with Venn diagrams have been implemented in Alice. Scratch has been used for algorithms and data representation (e.g. [Mal07]). Greenfoot (in greenroom.greenfoot.org) includes a number of exercises based around Computer Science concepts. In general, however, initial learning environments are used to encourage students to explore programming, not solve typical Computer Science problems.

CASE STUDIES
We implemented a parallel sorting algorithm, a sorting algorithm and a binary number conversion game in Scratch and in Alice. We chose three Computer Science Unplugged activities because relatively young students could understand these topics in non-programming activities. We wanted topics both accessible to beginning programming students via an unplugged-like presentation, and that required programming skills at the level of help provided online with each language. Although we trialled these activities with some novice programmers, this paper is based on an analysis of the language features.

The first case study was a parallel sorting network. In the Unplugged activity, children move through the network, and when two arrive at a node they compare their data values, with the smaller one taking the path to the left and the other to the
right. After going through the network, they come out in sorted order. This activity requires key comparisons, which are common in many fundamental algorithms, but also makes use of the animation that is natural in Scratch and Alice, to emulate the Unplugged version where children physically walk through the network.

The second concept was general sorting algorithms; we chose selection sort. This concept needed the languages to be able to easily use arrays and their basic functionality, and to show the operations on the array visually. The final concept was converting binary numbers to decimals, which requires working with bits rather than integers.

While we used Scratch and Alice, our methodology for evaluating other visualization based programming environments and languages designed for educational purposes (such as Greenfoot [Hen04] and DoLittle [Kan04]) would be similar.

CS CONCEPTS IN ALICE
The Alice programming language allows users to easily create 3D virtual worlds using a “drag and drop” programming language. We used Version 2.2, release 4/30/09 (Windows version). Alice aims to be a first-exposure into object-oriented languages, but it protects students from considerations such as syntax or complex event binding.

Version 3 of Alice is currently available as a beta version. While it does improve on some features (including providing the “Sims” as characters for animation), at the time of our research it was missing some features, such as the visualization objects that we found useful in our case study. Although Alice 2.2 and 3 can export code to Java and be programmed in Java, we focused on the novice programming environment.

Sorting Network in Alice
To implement a sorting network, we created a sorting network and three animals who start at the squares on the left (Figure 1). When the animals meet at the double circle, they compare their values, with the smaller one following the line to the left, and the larger one following the line to the right. Unfortunately, Alice does not provide a means to easily compare two objects based on their proximity, so creating a general-purpose implementation would require auxiliary data structures that would make the implementation difficult for beginner programmers. Instead we implemented a “hard-wired” version, which animated the moves for a particular choice of input values. We did this by using a set of objects to create the network on the ground, then using other objects as characters to move along the lines using the various “move to” functions. A student using this approach would still need to understand the general principle, but they would only be implementing a specific instance.

Figure 1: A sorting network in Alice

Another approach is to replace the start and end nodes with object visualization objects, and the intermediate nodes with array visualizations. Then, the steps may be automated without significant work, but the code would be harder for beginners to comprehend.
Figure 2: Referencing a variable in Alice

One problem encountered with the sorting network is the difficulty comparing objects by a property other than a built-in one, e.g., height. Alice provides a function, shown in Figure 2, to reference a variable in an object. In this case, a generic “animal” object is being referenced for its “myNumber” variable, of the Number type. Achieving a general approach requires sufficient compromise in the elegance of the programming that students are likely to miss the point of the exercise and be caught up working around the language.

Sorting Algorithm in Alice
Alice provides an easy way to show the items in an array, and they can be animated by using an object called the “array visualization” (see Figure 3). This object works much like an array, but has special methods for changing the array state that also show the change visually. This greatly simplified the amount of code needed, meaning the algorithm code for sorting was much the same as a traditional selection sort algorithm (see Figure 4). The value being compared was the built-in property of height.

Binary Numbers in Alice
Creating a binary converter in Alice could be done simply by using the “when I click anything” event, which returns the object clicked on to be used as a parameter in the code. By creating several objects (cubes) to represent the bits, and using this event, it is easy to simulate a “flip” event when each bit is clicked on, which changes its color between black and white (see Figure 5). Having each bit in an array allows for easy calculation of what value each bit represents. One problem with this implementation is that clicking on the sky causes an exception error in Alice.

General issues in Alice
Although it was possible to demonstrate all the CS concepts in our case study using Alice, there were some issues: In Alice version 2, each object is saved uniquely, even multiple identical objects, causing long save times. Thus, the size of the sorting array had to be limited. Alice can handle a few hundred simple objects, but with more complex objects, such as a penguin (with multiple parts and extra methods), numbers closer to 100 caused these issues to become rather significant. This is unfortunate, as the $O(n^2)$ vs.
0(nlogn) behavior of sorting algorithms becomes more obvious as n becomes large. Alice may even crash saving these larger programs, losing the program altogether. Version 3 of Alice fixes this problem, with saved files being both much smaller in size and without significant save times.

Another problem in Alice arises when trying to display a large number of items. It would be difficult to show an array of 100 items at a distance that allows an overview of the full array. This problem can be worked around using camera movement, a 2D view, or having characters being compared highlighted, but doing this also increases the complexity of an otherwise simple task. Another minor issue can be seen in Figure 5. The number variable in Alice adds a decimal to numbers. This is distracting because it implies that the number is not an integer.

Alice also has a few other minor problems that can create additional work for users, including not being able to make arrays smaller once a larger array is created, no way to delete user-created methods and functions, and objects such as the list visualization having bugs in their functionality. In these cases, a user may have to recreate their work rather than simply removing parts of the program. Alice’s drag and drop environment also can be fussy about what may be added to a particular statement. These issues are likely to be removed in version 3, and even with them present, we were able to implement all three concepts without using complex work-arounds, indicating that a novice programmer could work with them in this animated environment given suitable guidance.

**CS CONCEPTS IN SCRATCH**

Scratch is a 2D virtual world programming language from MIT Media Lab, aimed at helping young children develop programming skills. The language is based on programming sprites (typically characters or moving objects), which interact on a fixed-size background, and have “costumes”, which can be used to change a sprite’s appearance, or used with a timing script to create basic animations. These costumes can also be imported from existing images, and exported for use in other Scratch programs. Much like Alice, it uses a drag and drop structure with pre-defined code blocks to allow users to create programs without worrying about syntax. “Jigsaw puzzle” shapes enforce syntax by showing what kind of construct can be placed in any particular gap. We used Scratch 1.3.1 during this case study. Version 1.4 would not affect our conclusions.

Programs made in Scratch can also be easily uploaded to the Scratch website online. This facility allows easy sharing of programs, allows students to download and modify existing programs, and allows users to see the programs running through their web browser.

**Sorting Network in Scratch**

As with Alice, the sorting network was done as a hard-wired approach in this study, as Scratch does not have the functionality to easily build methods such as a “compare” method that can take sprites or their relevant variables as an input. The hard-wired approach is rather simple to implement because, similar to Alice, each sprite in Scratch can use the “go to” instruction to move
directly to the sprites representing the board (Figure 6).

**Figure 6: A multiple sprite Scratch sorting network**

**Sorting Algorithm in Scratch**

Scratch lacked the features we used in Alice to create a simple visual sort of an array. Scratch does not currently allow for its list functionality to have lists of sprites, which makes it difficult to easily compare and swap two sprites. It is, however, possible to compare two numbers in an array and swap them with a basic textual representation of the array being shown. While this is not as visual as Alice’s array visualization, it does allow for simple text-based viewing of the array, and can be used to show step-by-step changes in the array structure. The resulting sorting algorithm is similar to what might be done in a conventional language, sorting arrays of numbers, and displaying the numbers at each step.

**Binary Numbers in Scratch**

The Scratch web site already has binary number converter projects (e.g., http://scratch.mit.edu/tags/view/binary). So, we extended the idea of bit manipulation by implementing two variations of the Parity game from the CS Unplugged collection. The parity error correction represents a simple rule that has significant implications for data storage and transmission – an extra bit is added to each row and column to make up the number of 1 bits to an even number, enabling changed bits to be detected and corrected later.

Figure 7 shows the parity game implementation. The program chooses random gray and white bits. The users must click on the parity (check) bits to make sure there is an even number of gray bits in each row. When they click on the computer icon the number of correct and incorrect parity bits are displayed (8 incorrect, 3 correct in this case). We designed another game to demonstrate correcting an error.

**Figure 7: A parity-setting game in Scratch**

Bits in this instance are represented using two separate “costumes” which are switched upon click by a simple event. Variables keep track of the state of each bit, and thus what the parity bit should be. Only a few basic scripts needed to be written, which could then be copied between the objects. The problem with this is that if code requires editing, all the sprites would need to be edited individually.

**General issues in Scratch**

Scratch presented some challenges when implementing these problems. For the Sorting Network (Figure 7), the size of the world is simply too small for larger networks or substantial array visualization, due to the combination of both the large size default size of sprites and a small maximum size for the world.

Scratch also lacks the means for working with a group of sprites in an array or list construct. Many Computer Science concepts could be more easily
visualized if a number of sprites could be iterated through. It also means that a significant amount of redundant code is produced. Still, we were able to implement all three concepts in a way that a novice programmer could be guided through. In fact, binary number conversion had already been implemented by students.

CONCLUSIONS
Scratch and Alice represent a creative and engaging way to teach programming, and we were able to use them to program significant computer science concepts. The array visualization object in Alice is one feature that helps reduce the extra load that occurs when adding visual aspects. While less stable than Scratch, Alice does provide more features similar to a general-purpose language such as Java or VB, although it is missing a number of key language features, such as true class structures and inheritance, that these languages have.

Scratch provides an easy-to-use environment, but lacking features such as sprite arrays, simple parameter passing, and a larger virtual world size, it can restrict what can be implemented, and how it can be implemented.

While we didn’t evaluate the Greenfoot environment specifically, it clearly avoids some of the restrictions of Alice and Scratch because it has full Java programming available behind it. Indeed, a number of activities related to the ones discussed here have been published in greenroom.greenfoot.org. It is encouraging that the languages are evolving, and that the authors of Alice, Scratch and Greenfoot are discussing the capabilities of their environment with the Computer Science education community [Fin10].

ACKNOWLEDGMENTS
We are grateful to the students and novice programmers who have allowed us to look at their programs, and gain insight into a child’s experience of programming.

REFERENCES