Developing Children’s Computational Thinking using Programming Games

Simon Rose

A thesis submitted in partial fulfilment of the requirements of Sheffield Hallam University for the degree of Doctor of Philosophy

October 2019

Abstract

Computation is a fundamental part of our world, with today's children growing up surrounded by technology. This has led governments and policymakers to introduce computer science into primary and secondary education (age 5 to 16). These developments have been driven by 'computational thinking': the idea that the problem-solving skills used in computer science are useful in other disciplines. They have resulted in a wide range of programming tools designed for novices, of which Scratch, a block-based visual programming environment, is the most popular. Yet, so far, both computer science education and claims of computational thinking as a universal skill have failed to live up to their potential.

This thesis begins by reviewing the literature on computer science in primary education and computational thinking. It then describes a study that aimed to reproduce findings that programming improves story sequencing, a non-computational skill, in young children (age 5 and 6) using a programming game. The results showed an overall improvement for both the intervention and control group. In addition, it highlighted issues with teaching programming to young children. The thesis then refocuses on teaching older children (age 9 to 11) the computer science skill of abstraction and the idea that it can be used to refactor code to remove 'code smells' (bad programming practices). Code smells indicate an underlying problem in a program, such as code duplication, and are common in Scratch projects. A study is then reported that establishes that primary school children can recognise the benefits of abstraction when asked to alter Scratch projects that contain it.

The thesis then describes the design and development of Pirate Plunder, a novel educational block-based programming game designed to teach children to use abstraction in Scratch, using custom blocks (parameterised procedures) and cloning (instances of sprites). Two studies are reported in the subsequent chapters. The first investigates the value of a debugging-first approach in Pirate Plunder, finding that it was not always beneficial. The second measures for improvements in using abstraction in Scratch, finding that children who played the game were then able to use custom blocks to reduce duplication code smells in a Scratch project. In addition, Pirate Plunder players improved on a computational thinking assessment compared to the non-programming control group. The final chapter discusses the original contributions of the thesis, the implications of these and future direction.

Declaration

I hereby declare that:

1. I have not been enrolled for another award of the University, or other academic or professional organisation, whilst undertaking my research degree.

2. None of the material contained in the thesis has been used in any other submission for an academic award.

3. I am aware of and understand the University's policy on plagiarism and certify that this thesis is my own work. The use of all published or other sources of material consulted have been properly and fully acknowledged.

4. The work undertaken towards the thesis has been conducted in accordance with the SHU Principles of Integrity in Research and the SHU Research Ethics Policy.

5. The word count of the thesis is 56,656.

Name	Simon Rose
Date	October 2019
Award	PhD
Faculty	Science, Technology and Arts
Director of Studies	Dr Jacob Habgood

Publications

Rose, S. P. (2016). Bricolage Programming and Problem Solving Ability in Young Children: An Exploratory Study. *In Proceedings of the 10th European Conference for Game Based Learning* (pp. 914–921).

Rose, S. P., Habgood, M. P. J., & Jay, T. (2017). An Exploration of the Role of Visual Programming Tools in the Development of Young Children's Computational Thinking. *Electronic Journal of E-Learning*, 15(4), 297–309.

Rose, S. P., Habgood, M. P. J., & Jay, T. (2018). Pirate Plunder: Game-Based Computational Thinking Using Scratch Blocks. *In Proceedings of the 12th European Conference for Game Based Learning* (pp. 556–564).

Rose, S. P., Habgood, M. P. J., & Jay, T. (2019). Using Pirate Plunder to Develop Children's Abstraction Skills in Scratch. *In Proceedings of the Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*.

Acknowledgements

It’s fair to say that this thesis may never have been written if it hadn’t of been for Jacob Habgood, who took a promising software engineer and turned him into a (reasonably) competent researcher. Credit must also go to my other supervisor, Tim Jay, who provided complementary experience and expertise. Thanks to both for their guidance and support.

Thanks to my parents, Tim Rose and Liz Rose, for their unconditional love and support, and a special mention to my Dad’s teaching history and his current job that gave me a foot in the door with several of the schools involved.

A huge thanks to the children at Carterknowle Primary School, Hallam Primary School, Hillsborough Primary School, Meynell Primary School, Southey Green Primary School, Wisewood Primary School and Woodthorpe Primary School that took part in the game testing and the studies, it wouldn’t have been much of a PhD without them!

Many thanks to all the teaching staff involved who have been wonderfully helpful and accommodating. A special mention to Vicky Harris, Hannah Gleadhall, Mary-Jane Wright, Chloe Rudkin and Gill Clarke. Finally, saving the most important contribution until last, to Daniel Dwight, Ruth Kingdon and Bekki Lee who were all instrumental in ensuring that the two largest studies ran smoothly, and to Chris Stewart for giving them the go-ahead.

Thanks to my friends and family for keeping me grounded throughout the PhD and providing a much-needed distraction at times.

Finally, thanks to the creators of the free assets used in Pirate Plunder and to Caleb O’Brien for his valuable artistic contribution to the story sequences.

Dedication

To Liam Buckley, who would no doubt have joined me in being a student as long as possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn."

— Stephen Hawking

The debate over whether computer science should be taught in compulsory education has become more important as technology and computing become ubiquitous in society. Computation is a fundamental part of our world, with today's children growing up surrounded by technology and entering a job market that requires an ever-increasing number of computer-literate professionals (Passey, 2017). Over recent years, this has led governments worldwide to evaluate the teaching of computing in compulsory education (Heintz, Mannila, & Farnqvist, 2016). The result has been a shake-up of computing education, with previous curricula based on office software replaced with wider topics of informatics, digital literacy and computer science (e.g. The Royal Society, 2012). These developments have, in part, been driven by the idea of 'computational thinking', that the ideas and problem-solving skills used computer science can be useful in other disciplines (Wing, 2006). The term has subsequently been used by governments and policymakers as justification for teaching computer science in primary and secondary education.

Yet, despite the promise of computer science in primary education, it has so far failed to live up to its potential (The Royal Society, 2017). Teaching is often sparse and inconsistent, stemming from a lack of adequate training programs, infrastructure and materials. It is often left up to inexperienced teachers to develop an understanding of the required learning content by themselves. The multitude of programming tools, differing in type, cost, complexity and learning approach, means that educators are unsure which of these they should be using. This results in inconsistencies from school to school on how learning content is delivered, if at all. These problems are exacerbated by the lack of standardised computer science assessment in primary education. Despite the importance placed on it by governments, computer science often falls behind the traditional subjects of mathematics, literacy and science in primary school priorities.

This thesis is concerned with the way that programming tools are used to teach computer science in primary education. The main aim is to use computer science and computational thinking research to create a programming game that can be used in primary schools to effectively teach programming without prior knowledge on the part of the child or the teacher. Questions are raised throughout the thesis on computer science learning content, computational thinking and the efficacy of widely-used programming tools. These are then discussed in the final chapter (Chapter 10).

1.1 Aims

The aims of the thesis are to:

1. Review the literature on computer science education and computational thinking.
2. Identify areas of weakness in computer science education.
3. Create a programming game that addresses these weaknesses using game-based learning research.
4. Evaluate the programming game in a series of experimental studies.

The thesis focuses on the teaching of abstraction skills to children as an area of weakness in computer science education. Specifically, the concept of code reuse, which makes programs easier to understand and maintain. Code reuse is best achieved in simple programs through the extract method: moving fragments of duplicated code into a procedure that can then be called from multiple places within the program, meaning that the code fragment only exists in a single location (Chapter 5). This process is otherwise known as procedural abstraction and is notoriously difficult for novice programmers to learn, even in higher education (Kallia & Sentance, 2017).

Pirate Plunder, a programming game, is designed to teach these skills using Scratch's block-based programming language. Scratch (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) is the most widely-used programming tool in primary education. Yet, Scratch users frequently produce bad programming habits and code smells that make programs difficult to understand, debug and maintain (Aivaloglou & Hermans, 2016). These problems can be avoided using abstraction.

1.2 Thesis Structure

Chapter 2 analyses the growing trend for teaching computer science in primary education (age 5 to 11). The motivation behind this is that children should understand how technology works, to produce programmers for the growing technology industry and to foster logical thinking and problem-solving skills. Despite the evidence that primary-age children can develop programming skills, there are issues with the implementation of current curricula. The chapter explores how computer science has been introduced into compulsory education worldwide and looks at some of the available educational programming tools and their approach to teaching computer science.

Chapter 3 focuses on 'computational thinking': the logical thinking and problem-solving concepts used when solving computational problems. These concepts include abstraction, generalisation, algorithms, decomposition and debugging amongst others. Unlike computer science, computational thinking, in its current state, is a relatively new field of research. Some proponents of computational thinking argue that it is a 'universal skill' that should be part of non-computational disciplines, such as mathematics and science. However, there is a counterargument that computational thinking should be used as an explanation of the benefits of computer science, not as a discipline in its own right. The chapter explores this debate, examining definitions of computational thinking and computational thinking measures, before discussing the criticisms of computational thinking, including scope, transfer and teaching.

Chapter 4 reports a study designed to test the results of previous research into programming and its impact on non-computational skills. In a series of studies, Kazakoff, Sullivan & Bers (2012, 2014; 2013) found that a programming intervention improved the story sequencing ability of children age 4 to 7. The study reported in this chapter aims to replicate these results using an active control group and a programming game, Lightbot Jr, for the intervention. The study found no difference between groups and explores the reasons for this. The chapter ends by charting a change of direction in the thesis, moving away from measuring computational thinking onto teaching computer science and software engineering principles to older primary school children (age 9 to 11).

Chapter 5 describes abstraction and its importance in computer science and software engineering. Novice programmers develop abstraction skills as they gain expertise (Lister, 2011). Research suggests that primary school children can develop abstraction skills, but there is limited evidence on the topic. The chapter explores issues with Scratch, the most popular programming tool in primary education. In particular, the prevalence of code smells such as duplicated code and long scripts that can be removed using abstraction.

Chapter 6 then describes a study designed to see whether children age 10 and 11 with limited Scratch experience can recognise the benefits of abstraction in Scratch. They were asked to compare projects that met the same outcome but used different levels of abstraction, as measured by Dr. Scratch, an assessment tool that analyses Scratch projects for evidence of different computational thinking skills. The results were positive, suggesting that children could recognise why custom blocks and cloning were useful, but only when asked to alter projects themselves.

Chapter 7 explains the design and development of Pirate Plunder, a novel educational programming game designed to teach children to use abstraction in Scratch. The chapter goes into detail on the different aspects of the game design, describing how it introduces abstraction in a way that rationalises and explains its use, how it fosters transfer to Scratch, how it allows minimal teacher instruction and interaction and how it motivates the player.

Chapter 8 describes a study to investigate whether using a debugging-first approach in Pirate Plunder is beneficial to players. The debugging-first strategy comes from the theory that novice programmers learn better when completing existing code than starting with an empty program (Van Merriënboer & De Croock, 1992). Two versions of Pirate Plunder, debugging-first and non-debugging, were compared to a Scratch curriculum using several assessment tasks with children age 10 and 11. In addition to measuring for differences between the Pirate Plunder versions, a Scratch assessment was used to see whether the participants could use the abstraction skills they had learnt in Pirate Plunder in a separate Scratch project. The Pirate Plunder results showed that the debugging-first version was no more beneficial than non-debugging. There was also no difference in the amount of abstraction used by the intervention group and active control groups on the Scratch assessment. However, during artifact-based interviews, participants who played Pirate Plunder could explain abstraction (using custom blocks) and how they would use it in the assessment.

Chapter 9 builds on the results and observations from the previous chapter, describing a study designed to evaluate whether Pirate Plunder can be used to teach primary school children (age 10 and 11) to use abstraction in Scratch. It used an updated (single) version of Pirate Plunder, revised assessment tasks and a partial-crossover design. The study compared Pirate Plunder to a non-programming spreadsheets curriculum (using an additional Scratch control group in the first phase). In addition to abstraction assessments, the study used a computational thinking assessment to measure improvements in computational thinking after playing Pirate Plunder. The results were positive, showing that children were able to use abstraction to reduce block and sprite duplication in Scratch and an improvement on the computational thinking assessment in comparison with the non-programming control group.

Chapter 10 concludes the thesis by summarising the original contributions, exploring the implications of these and the impact of results on the wider context of computer science education. The chapter finishes with an exploration of future direction.

1.3 Notes

1.3.1 Thesis Length

As the thesis is interdisciplinary, it is between the length of standard computer science thesis and a standard education thesis. This is because it focuses on the educational and learning sciences aspects of the research, particularly on the learning outcomes of the participants in the four experimental studies reported. The thesis does not discuss the technical aspects of the software development as it is not the focus of the work.

1.3.2 Pirate Plunder

The name ‘Pirate Plunder’ in this thesis refers to the game created by the author and should not be confused with the programming game of the same name produced by Delightex, ‘Plunder Pirates’ developed by Rovio Entertainment or ‘Pirate’s Plunder’ by Dexterity Software. The name will be changed should it be released commercially in future.

1.3.3 Terminology

Throughout the thesis, the terms ‘function’, ‘procedure’ and ‘method’ are used interchangeably to refer to programmatic subroutines: a sequence of instructions that performs a specific task that is referenced within a larger body of code. There is a technical difference between these terms in that functions can return a value, procedures do not, and methods are normally associated with an object in object-oriented languages. However, this distinction is not relevant to this thesis.

1.3.4 Design Concepts

Several design concepts are highlighted using grey boxes such as this, which feed into the design and development of Pirate Plunder. This is done to aid the reader’s understanding of how the literature review influenced the game design.

1.3.5 Ethical Approval

Initial ethical approval was given by the university for Studies 1 and 2 (Appendix C). A change in the University ethics approval system (from document submissions to an online portal) meant that an amendment to this approval for Studies 3 and 4 (Appendix G) had to be submitted as a separate application.

Chapter 2

Computer Science in Primary Education

The lives of today's children will be greatly influenced by computing, both in the home and at work (Barr & Stephenson, 2011). Policymakers, supported by the technology industry, are arguing for children to be taught how technology works, to produce 'digital citizens' for an increasingly IT-based global economy. This has led to many countries introducing computer science (CS) into primary education (age 5 to 11) and an increase in the number of programming tools available for novice programmers.

This chapter covers the reasons for teaching CS in primary schools, its current state in countries around the world and an analysis of educational programming tools.

2.1 The Promise of Computer Science Education

Programming is becoming an increasingly important skill as technology becomes more prevalent in society. Nearly everyone in countries with advanced economies uses technology on a daily basis. Children are now able to use smartphones and tablets as young as 3 or 4 years old, often before they can read (Calvert, 2015). There is little doubt that today's children will interact with technology throughout their working lives, regardless of career choice.

Countries are under pressure to produce computer-literate professionals as the global economy becomes more driven by technology. However, influential government-funded reports in the UK (Livingstone & Hope, 2011; The Royal Society, 2012) and the US (A. Wilson & Moffat, 2010) have criticised the lack of adequate CS education at primary and secondary level (age 5 to 16). Since then, a combination of these reports, cheap and available technology and the growing software development industry (U.S. Department of Labor, 2018) have meant that traditional views that CS should only be taught in higher education are being reconsidered (Bocconi, Chioccariello, Dettori, Ferrari, & Engelhardt, 2016). Earlier teaching may also increase the number of female students studying CS at tertiary level and going on to work in the industry

(Margolis & Fisher, 2003), going some way to reducing the gender imbalance in the technology sector.

There is a growing consensus amongst researchers, policymakers and the technology industry for teaching children CS at an earlier age, in addition to the broader topic of ICT (Information and communications technology). In recent years, several countries have introduced CS into compulsory (primary and secondary) education (Heintz et al., 2016). There have also been attempts to introduce it outside the classroom, through code clubs and initiatives such as Hour of Code (C. Wilson, 2015). These developments have, in part, been driven by the 'computational thinking' movement that has gained traction in recent years. This is covered in detail in Chapter 3.

Passey (2017) summarises the main arguments for teaching CS in compulsory education:

- **Economic argument** - Education should develop the skills that are most likely to support a future IT-based economy.
- **Organisational argument** - Large organisations increasingly require technology-literate individuals to support their systems.
- **Community argument** - Computing facilities are being used increasingly by 'communities' for social purposes, in addition to organisations and individuals.
- **Educational argument** - Due to the speed that technology is developing, learners need an awareness and understanding of how it should be used responsibly.
- **Learning argument** - Developing problem-solving, collaboration, creativity and logical thinking skills through CS.
- **Learner argument** - Engaging learners in CS early, so that they have the opportunity to see how it might impact their future.

Conversely, there is a counter-argument that not all children need to learn CS: programming is unnecessary for most jobs, maybe automated or outsourced to developing economies in the future and can be too difficult for younger children to learn. Furthermore, suggestions that CS learning outcomes (e.g. logical thinking and problem-solving) can be transferred to other subject areas may be problematic (Denning, 2017) (Chapter 3). There are also issues with CS teaching because government changes to curricula have been made without adequate research or teacher training programs in place (Webb et al., 2017). CS in primary education is not part of standardised testing, so is given limited classroom time in comparison with traditional subjects. A follow-up report of the 2013 computing reforms in English compulsory education notes that whilst there are pockets of excellence, computing education overall is still "patchy and fragile" (The Royal Society, 2017, p. 6).

In summary, CS is being introduced into primary education to give children an understanding of how technology works, to produce programmers for a growing software engineering sector, to foster problem-solving and logical thinking skills and to give them an awareness of how it may influence their lives. The next section (Section 2.2) explores the countries that have introduced CS into compulsory education: how it has been integrated and what sort of content is taught.

2.2 The Introduction of Computer Science Education

Several countries have now integrated CS into their national curricula (Heintz et al., 2016). These changes are summarised in Table 2.1. CS has largely been given its own subject area, but some countries have integrated it into existing subjects such as mathematics. It is often taught as part of a broader informatics curriculum and is sometimes combined with developing 'digital competencies', such as using technology efficiently and responsibly, addressing issues such as e-safety and cyber-bullying. CS is compulsory at primary level (age 5 to 11) in just under half of the countries reviewed (9/20). Furthermore, stakeholders in several others, such as France and Spain, are pushing for its inclusion (Bocconi et al., 2016). With so much emphasis on CS, it is therefore important that learning content is suitable for primary school children and educational programming tools are well researched and used correctly.

2.2.1 What Children are Taught

Primary school children begin by learning to write simple programs using a variety of approaches: basic visual programming environments, games, physical devices and unplugged activities (e.g. writing out simple algorithms on paper). For example, the English national curriculum for computing at Key Stage 1 (age 5 to 7) (Department for Education, 2013) states that pupils should be taught to:

- Understand what algorithms are; how they are implemented as programs on digital devices; and that programs execute by following precise and unambiguous instructions.
- Create and debug simple programs.
- Use logical reasoning to predict the behaviour of simple programs.

Under 7 years old, children are often limited to low-level CS concepts like writing algorithms, sequencing instructions and developing basic prediction skills. This is primarily to introduce them to the basics, but also because of cognitive limitations in working memory (Van Merriënboer & Sweller, 2005) and abstract reasoning (Armoni, 2012). As they get older, children are introduced to selection, repetition and problem

Table 2.1: Overview of CS education in different countries, adapted from Heintz, Mannila, & Farnqvist (2016) using additional information from Bocconi, Chioccariello, Dettori, Ferrari, & Engelhardt (2016)

Country	What?	How?	Primary	Secondary
Australia	Digital Technologies	Own subject and integrated	Compulsory	Compulsory
Croatia	Informatics	Own subject	Elective	Compulsory
Cyprus	Computer Science	Own subject	-	Compulsory
Denmark	Informatics	Own subject	-	Compulsory
England	Computing	Replace existed subject	Compulsory	Compulsory
Estonia	Programming (Technology & innovation)	Integrated	Compulsory	Compulsory
Finland	Programming (Digital competence)	Integrated	Compulsory	-
Hungary	Information Technology	Own subject	-	Compulsory
Isreal	Computer Science	Own subject	-	Compulsory or elective depending on institution
Lithuania	Information Technology	Own subject	-	Compulsory
Malta	Digital Literacy	Integrated	Compulsory	Elective
New Zealand	Programming and Computer Science	Own subject	-	Elective
Norway	Programming	Own subject	-	Elective
Scotland	Computing science	Own subject	-	Elective
Slovakia	Informatics	Own subject	Compulsory	Compulsory
South Korea	Informatics	Own subject	Compulsory	Elective
Sweden	Programming and Digital Competence	Integrated	Compulsory	Elective
Poland	Computer Science	Own subject	Compulsory	Compulsory
Turkey	Computer Science	Own subject	-	Compulsory
USA	Computer Science	Own subject	-	Elective

decomposition, as shown in the English national curriculum for Key Stage 2 (age 7 to 11) where pupils are taught to:

- Design, write and debug programs that accomplish specific goals, including controlling or simulating physical systems; solve problems by decomposing them into smaller parts.
- Use sequence, selection, and repetition in programs; work with variables and various forms of input and output.
- Use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs.

2.2.2 What Children can Learn

The difficulties that children under 7 have with the more abstract CS concepts (e.g. selection, repetition, debugging, variables and procedures) can be explained by Piaget's stages of cognitive development (1970), which provide a framework for when children gain awareness of the world around them. In traditional Piagetian theory, children are not able to think abstractly and reason about hypothetical problems until they reach the 'formal operational' stage. This abstract reasoning is fundamental in CS and is "one of the most vital activities of a competent programmer" (Dijkstra, 1972, p. 864), which might explain why younger children can struggle to predict the outcome of programs, even if they understand the syntax.

Traditional Paigetian theory states that children do not reach the 'formal operational' stage until around 11 years old. However, neo-Piagetian theory suggests that "people, regardless of their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain" (Lister, 2011, p. 2), which is supported by Piaget's later work (2001). Lister goes on to describe novice programmers' behaviour using neo-Piagetian stages, with expert programmers being able to reason at the highest 'formal operational' stage:

Sensorimotor Stage - Struggle with syntax and require considerable effort to trace code (mentally simulate the program to predict program behaviour). Often manipulate code using trial and error.

Preoperational Stage - Understand basic programming concepts, so can more reliably trace code. Reliant on specific values to trace, understand and write code. Struggle with the abstract relationship between different parts of the code as their focus is limited to a single statement or expression at a time.

Concrete Operational Stage - Can reason at a more abstract level, not reliant on specific values. Able to understand short pieces of code simply by reading (tracing not required) and can perform transitive inference: comparing two objects via an intermediary object.

Table 2.2: Categorising programming responses using the SOLO taxonomy, adapted from Seiter (2015)

Category	Original Explanation	In Programming
Prestructural	The response contains bits of unconnected information, demonstrating a misconception of the task.	The code substantially lacks knowledge of programming constructs or is unrelated to the question.
Unistructural	The response focuses on one relevant aspect of the task. Progression to the next level is quantitative.	The code represents a direct translation of the specification, it is in the sequence of specifications.
Multistructural	The response focuses on several aspects, but relationships among aspects and their significance to the whole is missed. Progression to the next level is qualitative.	The code represents a translation that is close to direct. The code may have been reordered to make a more integrated, valid solution.
Relational	Meta-connections are made, and the response is holistic, integrating concepts into a coherent whole.	The code provides a well-structured program that removes all redundancy and has a clear logical structure. The specifications are integrated to form a logical whole.
Extended abstract	The response demonstrates conceptualization at a higher level of abstraction, formulating an instance of a general case.	The code uses constructs beyond those required in the exercise to provide an improved solution.

Another view comes from Seiter (2015), who used the SOLO taxonomy to classify programming responses of primary school children. SOLO classifies learning outcomes into five levels of increasing structural complexity, which Seiter revised for ‘code writing’ (both have been provided in Table 2.2). The SOLO taxonomy mirrors the progression through neo-Piagetian stages, with learners becoming better able to view code holistically and abstractly as they develop their skills.

Both neo-Piagetian theory and the SOLO taxonomy imply that the earlier children begin to develop expertise in CS, the faster they will be able to develop a holistic understanding of code and of more abstract programming principles, like selection, repetition, debugging, variables and procedures. As long as their working memory is sufficient. This is supported by several studies that indicate that primary school children can learn to understand abstract CS concepts using structured learning content (e.g. Gibson, 2012; Price & Price-Mohr, 2018).

Design Concept - Abstract Reasoning
Primary school children find abstract reasoning difficult but should be able to develop those skills in programming if the learning content is structured correctly.

2.3 The Reality of Computer Science Education

The previous section implies that children should be able to start learning CS concepts at around age 5. Yet, this is largely dependent on the teaching methods and programming tools used (Duncan, Bell, & Tanimoto, 2014; Shein, 2014). As mentioned briefly in Section 2.1, many primary school teachers are not appropriately trained to deliver CS content (Webb et al., 2017). This is a particular issue in England and Australia, where curriculum changes have been criticised as being made prematurely by the government at the risk of inadequate teacher knowledge affecting learning outcomes (Brown, Sentance, Crick, & Humphreys, 2014). Often teachers must either take an active interest in training themselves and finding appropriate curriculum content, or children are given programming tools to use with little or no guidance. In some cases, CS may even be ignored because of outdated equipment, lack of teacher confidence and testing pressure on traditional subjects like mathematics and science (Yadav, Gretter, Hambrusch, & Sands, 2016). The lack of teacher knowledge increases the importance of educational programming tools in meeting learning outcomes.

There are a wide range of educational programming tools designed for primary school children (Rich et al., 2019). The next section (Section 2.4) will give an overview of these tools before describing each category in more detail.

2.4 Educational Programming Tools

For this thesis, educational programming tools include any software, hardware or approach that uses programming languages or methods designed for novices. This covers visual programming environments (VPEs), programming games, physical devices and unplugged activities. It is worth noting that some of these tools have been developed by researchers and have some empirical support, yet the majority are commercial and lack published research.

This section gives an overview of available educational programming tools, a brief description of their history and an explanation of each category listed above with some example tools.

2.4.1 Available Tools

Table 2.3 gives an overview of the tools available. It has been extended from an analysis by Duncan, Bell, & Tanimoto (2014, p. 66) and uses their heuristics to classify each tool according to its approximate age group, ability level and learning outcomes:

> **Level 0** - Age 2 to 7. Drag-and-drop or simpler. Teaches planning (sequence) only. Requires no abstraction. Contains no significant use of: procedures, variables, iteration, indexed data structures, conditional execution.
>
> **Level 1** - Age 5 to 10. Drag-and-drop. Requires no abstraction (or small amounts). Contains none or few of: procedures, variables, iteration, indexed data structures, conditional execution.
>
> **Level 2** - Age 8 to 14. Drag-and-drop or text-based. Includes some abstraction. Contains some or most of: procedures, variables, iteration, indexed data structures, conditional execution.
>
> **Level 3** - Age 12 and above. Drag-and-drop or text-based. Includes abstraction. Contains all of: procedures, variables, iteration, indexed data structures, conditional execution.
>
> **Level 4** - Age 14 and above. Teaches an industry-level Turing-complete programming language. Advanced, with extensions available. Contains all of: procedures, variables, iteration, indexed data structures, conditional execution.

The table is sorted by these heuristics, with more complex tools nearer the bottom. Programming-style refers to what sort of language the user is programming in. Block-based languages often combine blocks and text, giving the user a selection of blocks that snap together to form scripts. Some of these tools use blocks with symbols to support younger users, which have been specified. ‘Commercial’ indicates whether or not the product costs money. Target age has been left blank if no information from the developer could be found. Note that this is not an exhaustive table but has been included to give an overview of the range of available educational programming tools.

Table 2.3: Educational programming tools for children, extended from Duncan, Bell, & Tanimoto (2014)

Name	Type	Programming Approach	Release Year	Developer	Commercial	Target Age	Level
Bee-bot app	Game	Block-based (no-text)	2012	TTS Group	Yes	4+	0
Bee-bot	Physical device	Direct input	2008	TTS Group	Yes		0
ScratchJr	VPE	Block-based (no text)	2014	DevTech and MIT Media Lab	No	5-7	0 1
Crabs and Turtles	Unplugged	Block-based (no text)	2017	Tsarava et al. (2018)	No	8-9	0 1
Cork the Volcano	Game/Physical device	Block-based (no text)	2016	Digital Dream Labs	Yes	6-8	0 1
Lightbot Jr	Game	Block-based (no text)	2014	SpriteBox LLC	Yes	4-8	0 1 2
Box Island	Game	Block-based (some text)	2015	Radiant Games	Yes	6+	0 1 2
Lightbot	Game	Block-based (no text)	2008	Lightbot	Yes	9+	0 1 2
BOTS	Game	Block-based	2015	Liu et al. (2017)	No	11+	0 1
KIBO	Physical Device	Block-based	2017	KinderLab Robotics	Yes	4 - 7	0 1
Dash and Dot	Physical device/Game/VPE	Block-based (some levels with no text)	2016	Wonder Workshop	Yes	5-11	0 1 2 3

Continued on next page

Continued from previous page

Name	Type	Programming-style	Release year	Developer	Commercial	Target age	Level
Code.org	VPE/Game	Block-based (text and no text) and text-based	2013	Code.org	No	5+	0 1 2 3 4
Kodable	Game	Block-based (no text)	2012	SurfScore	Yes	4-11	1
Kodetu	Game	Block-based	2014	Deusto Learning Lab	No	8-15	1 2
Kodu Game Lab	VPE	Block-based	2009	Microsoft Fuse Lab	Yes	9+	1 2
ctGameStudio	Game	Block-based	2016	Collide Research Group	No		1 2
Cargo-Bot	Game	Block-based	2016	Two Lives Left	Yes		1 2
Dragon Architect	Game	Block-based	2015	Bauer et al. (2015)	No		1 2
Turtle Academy	VPE	Text-based (Logo)	2011	TurtleAcademy	No	7-12	1 2
Turtle Logo	VPE	Text-based	1967	Papert et al. (2011)	No	4+	1 2
Scratch	VPE	Block-based	2007	MIT Media Lab	No	8+	1 2 3
Tynker	VPE	Block-based	2013	Neuron Fuel	Yes	7+	1 2 3
Hopscotch	VPE	Block-based	2012	Hopscotch	Yes	8-12	1 2 3
Snap!	VPE	Block-based	2011	Jens Mönig	No	12+	1 2 3

Continued on next page

Continued from previous page

Name	Type	Programming-style	Release year	Developer	Commercial	Target age	Level
Gidget	Game	Text-based	2014	Lee & Ko (2014)	No		1 2 3
Code Avengers	Game/VPE	Block-based and text-based	2012	Online Education	Yes	5+	1 2 3 4
CodeHS	Game/VPE	Text-based	2012	CodeHS	Yes	5+	1 2 3 4
Alice	VPE	Block-based	1998	Carnegie Mellon University	No		2 3
CodeCombat	Game	Text-based (JavaScript)	2013	CodeCombat	Yes	9+	2 3
Stagecast Creator	VPE	Rule-based (text and image)	1998	Apple	Yes	8+	2 3
Looking Glass	VPE	Block-based	2013	Washington University in St. Louis	No	10+	2 3
Lego Mindstorms EV3	Physical device/VPE	Block-based or text-based	2013	Lego	Yes	10+	2 3 4
GameMaker	VPE/Game-engine	Block-based	1999	YoYo Games	Yes		2 3 4
App Inventor	VPE	Block-based and text-based	2010	Google/MIT	Yes		3 4
AgentSheets	VPE	Rule-based (text and blocks)	1991	Repenning (1992)	Yes		3 4

Continued on next page

Continued from previous page

Name	Type	Programming-style	Release year	Developer	Commercial	Target age	Level
Robot ON!	Game	Text-based (C++)	2016	Software Quality Research Lab (Miljanovic & Bradbury, 2016)	No		3 4

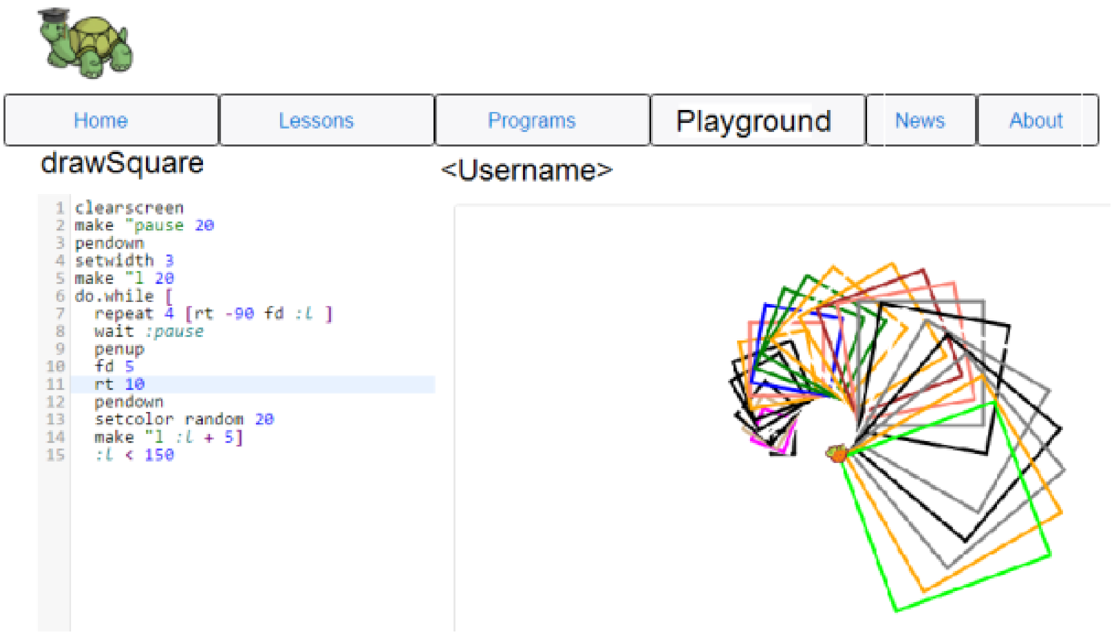
2.4.2 A Discussion of Available Tools

It is clear from Table 2.3 that there are a wide range of programming tools available for novice programmers. Block-based programming is the most popular approach, but some higher-level tools use text-based languages. Over half of the tools are commercial, showing that there are commercial opportunities in producing these tools. Most of the tools are specifically aimed at children. VPEs, games and physical devices are all used to teach at every level of ability as defined by the heuristics. The level and target age group of the tools differs slightly from the age range specified for each of Duncan, Bell, & Tanimoto's (2014) heuristics. Tools at levels 0 and 1 are aimed at children aged 4 and above, level 2 at age 8 and above, and levels 3 and 4 at age 10 and above. This means that tools that introduce abstraction are aimed at children aged 10 and above, fitting with the earlier discussion of what children can learn (Section 2.2.2). Primary school children can start with basic tools that allow sequencing and 'code tracing' tasks. They can then progress to using tools that contain abstract concepts like conditional execution, procedures and variables. However, there is a lack of research measuring and comparing the efficacy of these tools for primary school children (Fessakis, Gouli, & Mavroudi, 2013).

The rest of this section includes a brief description of the historical context and an explanation of each category and a selection of tools. As this thesis focuses on VPEs and games (specifically Scratch and Lightbot), more attention will be paid to these than to physical devices and unplugged activities.

2.4.3 A Brief History of Programming Tools

The Logo programming language was the first programming tool designed specifically for use in education (Feurzeig & Papert, 2011). Learners program a 'turtle' (represented by an on-screen cursor or a floor-robot) using simple text-based instructions. Procedural programming can be used to direct the turtle to draw complex geometric shapes (Figure 2.1). Seymour Papert was one of the first to see the potential of computing for learning (1980). He advocated that all children have access to a computer, believing that they should take control of their learning by using the materials around them and that both knowledge and problem-solving skills would come as a by-product of this exploration, a learning theory known as constructionism (Turkle & Papert, 1992). The work of Papert and colleagues led to computers, and Logo, being popular in schools throughout the 1980s. However, questions were raised over the effectiveness of Logo to develop general problem-solving skills (e.g. Kurland, Pea, Clement, & Mawby, 1986), which is discussed further in Chapter 3. The enthusiasm for programming in education faded before its revival in the 21st century, yet Logo-based environments and constructionism have inspired current programming tools, most notably Alice (Cooper, Dann, & Pausch, 2003), Scratch (Resnick et al., 2009) and Kodu Game Lab (MacLaurin, 2011).



```
clearscreen
make "pause 20
pendown
setwidth 3
make "l 20
do.while [
  repeat 4 [rt -90 fd :l ]
  wait :pause
  penup
  fd 5
  rt 10
  pendown
  setcolor random 20
  make "l :l + 5]
  :l < 150
```

Figure 2.1: Screenshot of a Logo program in Turtle Academy

2.4.4 Visual Programming Environments

Visual programming environments (VPEs) (sometimes referred to as visual programming languages or VPLs) can be used to create media, animations and games using specialised block-based programming languages. Blocks are used to implement programming concepts such as conditionals, loops, events and procedures. VPEs often follow an event-based approach, where blocks are executed when a specific event occurs.

Repenning (2017) describes three levels of features that make a successful VPE:

Syntax - Using blocks/icons, forms and diagrams to reduce or eliminate syntax errors and allow the learner to arrange well-formed programs.

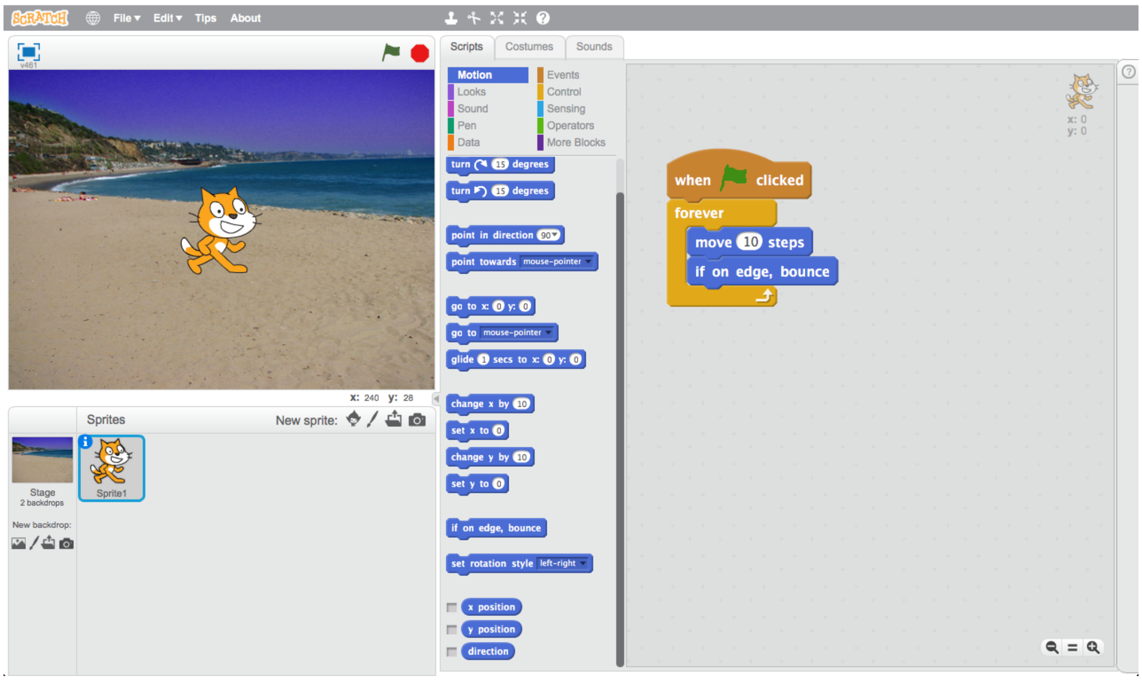
Semantics - Mechanisms to disclose the meaning of programming primitives. This is can be done through the shape of a block, e.g. a loop block having a gap to place other blocks inside it (as shown in Scratch in Figure 2.2), or through clear documentation.

Pragmatics - Give information on what a program means in a particular situation, e.g. how does it react when you add certain data or put it in a certain state. Stagecast Creator does this using 'rules' that allow the user to program behaviours based on what action triggers an event (e.g. hitting an obstacle) and what happens after the event (e.g. jumping over the obstacle).

Novices can find it easier to learn CS using block-based languages, over text-based languages, because they rely on recognition instead of recall (blocks are selected from a pallet), reduce cognitive load by chunking code into smaller numbers of meaningful elements and allow users to avoid basic errors by providing constrained direct manipulation of structure (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017). Most VPEs follow a ‘low-floor high-ceiling’ design approach that makes it easy for novices to get started but provides enough functionality for them to challenge themselves once they become proficient. Because of this low barrier of entry, they are increasingly used to teach programming in primary education. This section will describe three VPEs that are used in primary education and are relevant to the rest of the thesis; Scratch, ScratchJr and Kodu.

Scratch

Scratch (Figure 2.2) is one of the most popular VPEs, with over 37 million projects shared on its online platform since it’s public release in 2007 (Scratch Team, 2019). It is also the most taught environment in primary schools (Rich et al., 2019). Designed for children age 8 and above, it aims to “introduce programming to those with no previous programming experience" (Maloney, Resnick, & Rusk, 2010, p. 2).



```
when [flag] clicked
forever
    move 10 steps
    if on edge, bounce
```

Figure 2.2: Screenshot of a simple Scratch project

Scratch 2.0 has 116 blocks divided into 10 categories: motion, looks, sound, pen, data, events, control, sensing, operators and ‘more blocks’. Blocks are combined to form ‘scripts’ that are used to program ‘sprites’ to perform behaviours. Scripts are triggered by ‘event’ blocks, for example when a particular key is pressed or when a message sent from another script. Sprites are added to ‘backdrops’ to create stories, animations and games. Projects can easily be customised by importing or creating images, sounds and music. They can also be shared on the Scratch online platform and 'remixed' by other users (Dasgupta, Hale, Monroy-Hernández, & Hill, 2016). Scratch can be used to program robotics kits (e.g. Lego Mindstorms) and can take inputs from other physical devices like the Scratch Controller.

Scratch has been used from early years to higher education for teaching CS and as a stepping stone to text-based programming languages (Franklin et al., 2016). There is some evidence to suggest that Scratch can be used to improve wider skills of mathematics (Calao, Moreno-León, Correa, & Robles, 2015) and problem-solving (Giordano & Maiorana, 2014). Scratch includes abstraction and contains procedures, variables, iteration, indexed data structures and conditional execution (levels 1, 2 and 3).

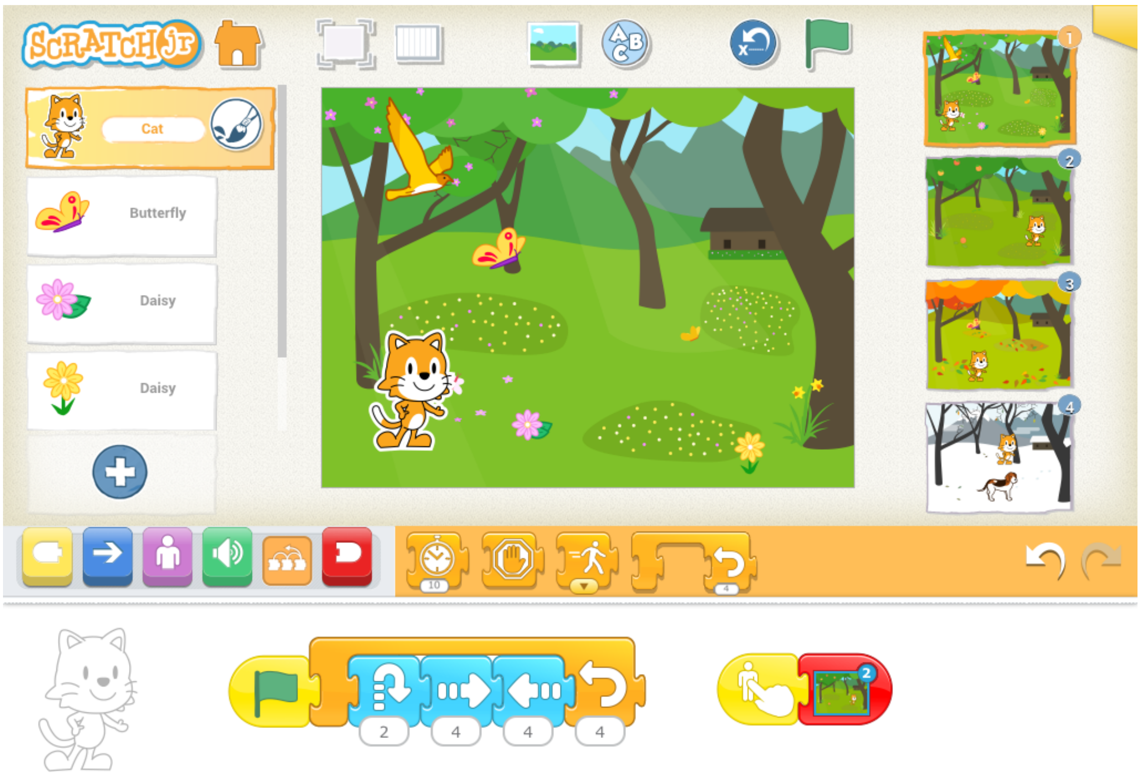


Figure 2.3: Screenshot of a simple ScratchJr project

ScratchJr

ScratchJr is a version of Scratch redesigned for younger children age 5 to 7. It maintains the creative programming elements of Scratch, allowing children to easily create short stories and games. It was developed using several constructionist age-appropriate design principles (Flannery et al., 2013) that are common in constructionism-inspired VPEs:

> **Low floor and appropriately high ceiling** - Easy to get started but providing room to use more complex concepts.

Wide walls - Many different pathways and styles of exploration.

Tinkerability - Ideas can be incrementally developed through experimentation.

Conviviality - The interface is friendly and playful.

Classroom support - Wide range of learning outcomes through:

- Feasible management of use in classroom settings.
- Support for building foundational knowledge which underlies multiple disciplines, such as sequencing, patterning and iteration.
- Support for discipline-specific knowledge from mathematics, literacy and classroom-selected criteria.
- Support for problem-solving strategies and skills.
- Complementary curricula and suggested teaching practices co-designed with early childhood teachers.

In ScratchJr, characters can be added to a scene and given behaviours by combining instruction blocks. The interface is entirely symbolic and contains only a third of the original Scratch instruction set. ScratchJr executes instructions from left to right (the way that the English language is read) instead of the top to bottom approach used in Scratch. It has large buttons for touchscreen use to alleviate difficulties that young children often have with mouse movement. Scratch's Cartesian coordinate system has been replaced by a natural coordinate system and there is a grid that can be overlaid on top of the scene to help children calculate distance. Numerical parameter values have a maximum limit of 25 and users can execute individual instructions simply by pressing on them, allowing them to explore what each instruction does.

Exploratory studies have found that ScratchJr can help young children familiarise themselves with basic programming concepts (Papadakis, Kalogiannakis, & Zaranis, 2016; Strawhacker, Lee, & Bers, 2017). ScratchJr does not include abstraction and contains only iteration and some conditional execution in the form of wait blocks (levels 0 and 1).

Kodu Game Lab

Kodu Game Lab (often referred to as Kodu) is a VPE developed by Microsoft and designed for children age 9 and above (Figure 2.4). It is integrated into a real-time 3D gaming environment that is used to create games through 'independent exploration' (MacLaurin, 2011). Users give behaviours to sprites when a certain event happens, for example when the player clicks on the sprite, or when the sprite hits a different sprite. All behaviours are built using a condition (when) and an action (do). Kodu supports flow control, boolean logic, simple use of variables (score) and inheritance (a sprite can be set as creatable and can then be created by other sprites) (Stolee & Fristoe, 2011). However, it is largely an explorative environment focused on seeing and moving rather than on abstractions like variables and procedures.

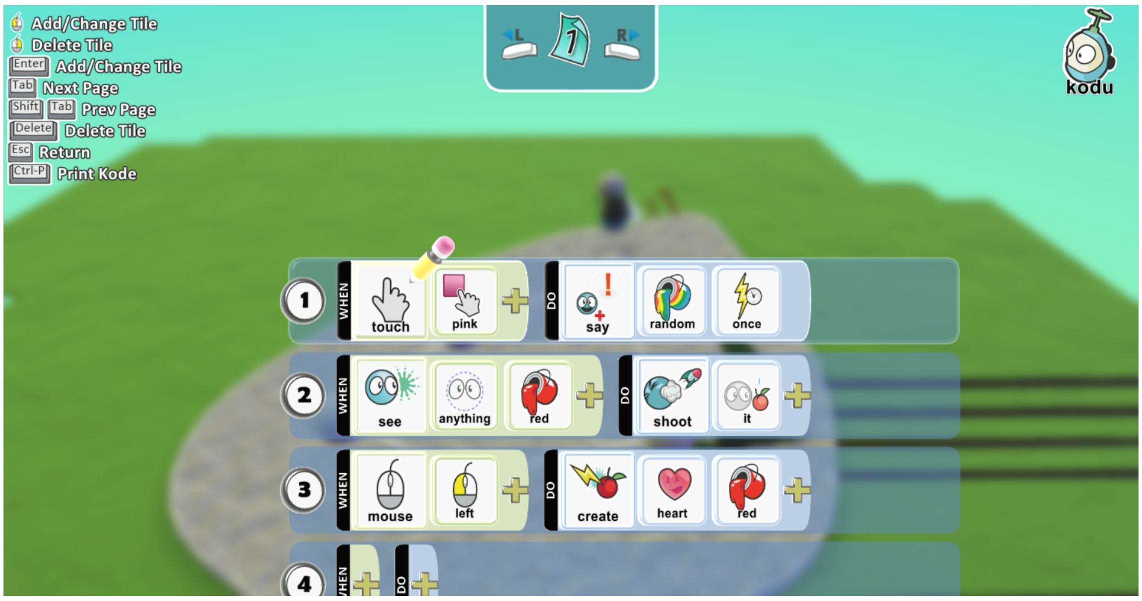


Figure 2.4: Screenshot of Kodu sprite programming

Some exploratory studies suggest that Kodu is good for enriching introductory programming experience (A. Fowler, 2012; Sovic, Jagust, & Sersic, 2014). Kodu includes some abstraction and contains limited use of variables, iteration and conditional execution (levels 1 and 2).

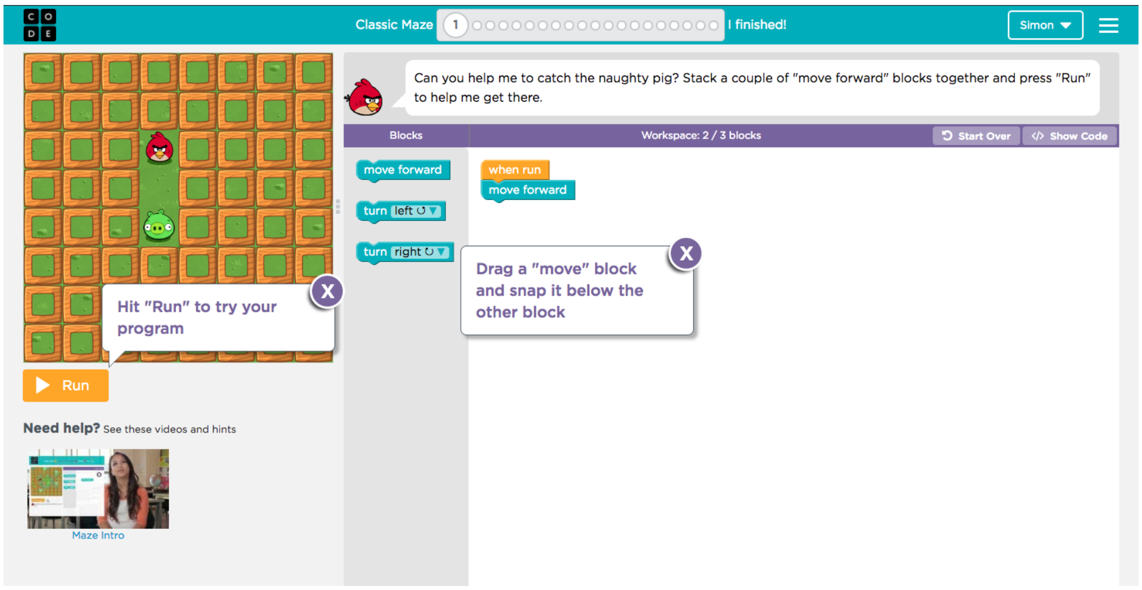
2.4.5 Programming Games

The benefits of game-based learning in educational contexts are well researched (Boyle et al., 2016). Programming games usually involve navigating an object through a grid, either using block-based or text-based instructions. Harms, et al. (2015) suggest that these puzzle-like approaches are more effective than tutorials for teaching programming to novices. This section will describe three programming games relevant to the rest of the thesis; Code.org, Lightbot and Lightbot Jr.

Design Concept - Combining Puzzles and Tutorials
Combining puzzle-based levels with tutorials that introduce programming concepts should allow a game to teach difficult content without external support.

Code.org

Code.org is a non-profit organisation dedicated to expanding CS access in schools. The initiative includes an online programming platform with sets of linear tutorials that teach programming constructs including algorithms, conditionals, variables, loops and procedures (Figure 2.5). Tutorials are sorted into 'courses' and 'lessons' that are categorised by age group. These also include unplugged activities (Section 2.4.8) to be used alongside the online platform. Tutorials aimed at younger children use a block-based language, whereas older children are introduced to the text-based languages JavaScript and Python (in some tutorials, users can switch between block and text-based code). The platform can also be used to create projects similar to Scratch in 'labs', with over 35 million projects created to date (Code.org, 2018). Code.org is supported by many large organisations including Google, Microsoft, ISTE and ACM and uses materials licensed from several well-known franchises including Angry Birds and Star Wars.


Can you help me to catch the naughty pig? Stack a couple of "move forward" blocks together and press "Run" to help me get there.

Drag a "move" block and snap it below the other block

Hit "Run" to try your program

Figure 2.5: Screenshot of the Classic Maze course in Code.org

Code.org begins with very simple programming for young children but does introduce abstraction in later sections and contains procedures, variables, iteration, indexed data structures and conditional execution (levels 0, 1, 2, 3 and 4). Kaleioğlu (2015) found that Code.org helped primary school children develop a positive attitude towards programming but did not improve their reflective thinking skills towards problem-solving.

Lightbot

Lightbot (sometimes stylised as Light-bot) (Figure 2.6) is a programming game designed for children age 9 and above. In the game, the player arranges a fixed set of block-based instructions to program a robot. Unlike Code.org and Scratch, each level has a limit to the size of the program the player can produce. The goal is to program the robot to 'light up' all the blue blocks on a level. This is done by navigating the robot to a blue block and executing the light command. Players can decompose a level into different sections, which can then be solved one after the other until they have a complete solution. Later levels introduce procedures and conditionals. For procedures, the player is given other program spaces below the main program that can be called using special instructions. Conditionals are implemented using a paint tool that colours the robot so that only instructions of that colour are executed.

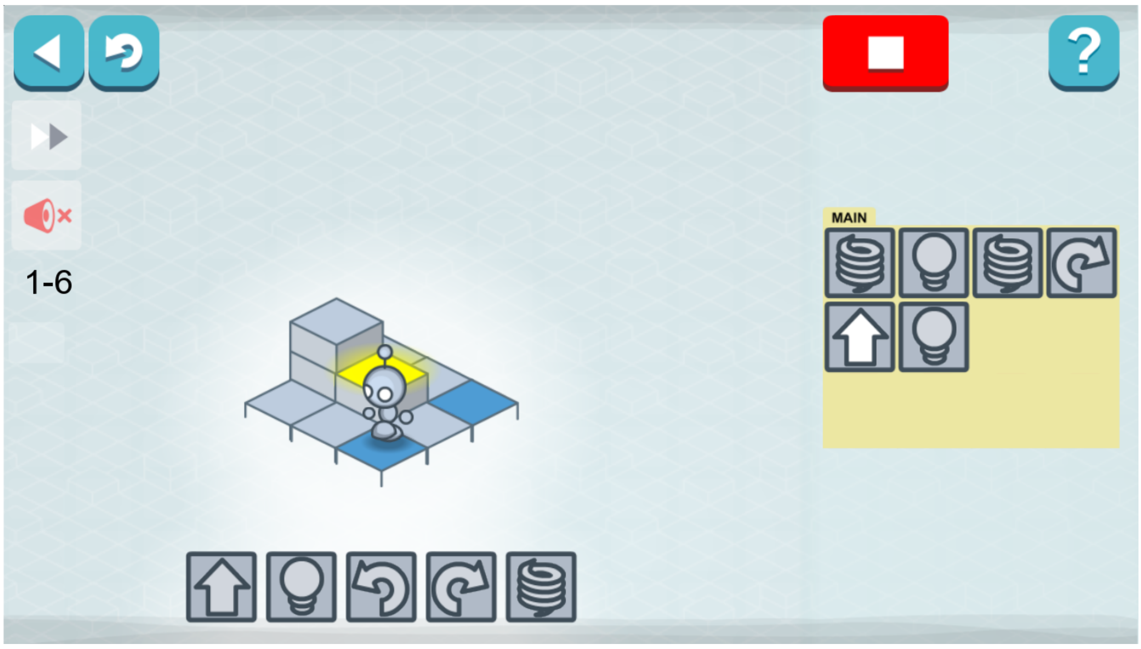


Figure 2.6: Screenshot of Lightbot

Gouws, Bradshaw & Wentworth (2013) suggest that Lightbot is useful for practising computational thinking as a problem-solving process, where players are rewarded for producing optimised solutions. Duncan, Bell & Tanimoto (2014) suggest that limiting available commands can force players into practising CS concepts like abstraction and decomposition. Lightbot includes some abstraction and contains procedures, iteration and conditional execution (levels 0, 1 and 2).

Design Concept - Limiting Available Commands
Programming concepts can be introduced to the player by limiting available commands and program space.

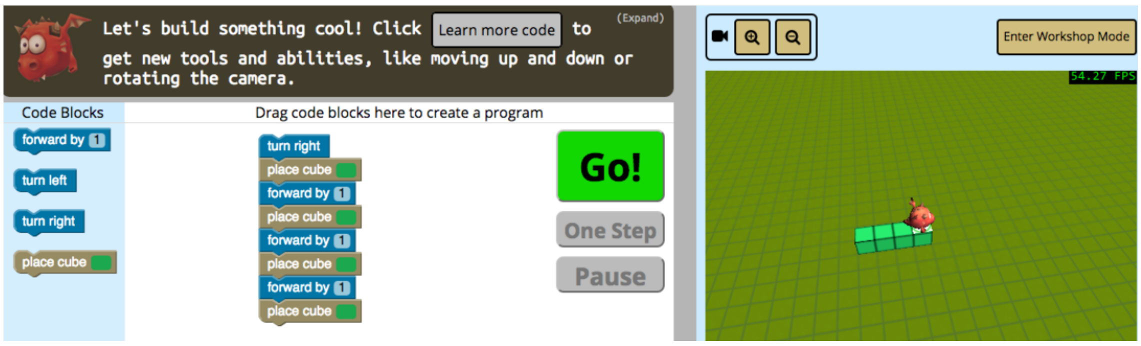
Lightbot Jr

Lightbot Jr is an educational puzzle game designed for children age 4 to 7. It is a version of Lightbot, where the levels have been simplified for younger children. Despite including some abstraction, the first two sets of levels concentrate on creating algorithms by predicting the outcome of the program. As with Lightbot, Lightbot Jr includes some abstraction and contains procedures, iteration and conditional execution (levels 0, 1 and 2).

2.4.6 Different Learning Approaches

Most programming tools sit on a scale between the open-ended exploration of VPEs like Scratch and Kodu, and linear puzzles with lots of direct guidance like Lightbot. Linear games can teach skills without external guidance by limiting player freedom and introducing concepts at a steady pace. The downside is that once a player has completed the game, they are unable to continue exploring.

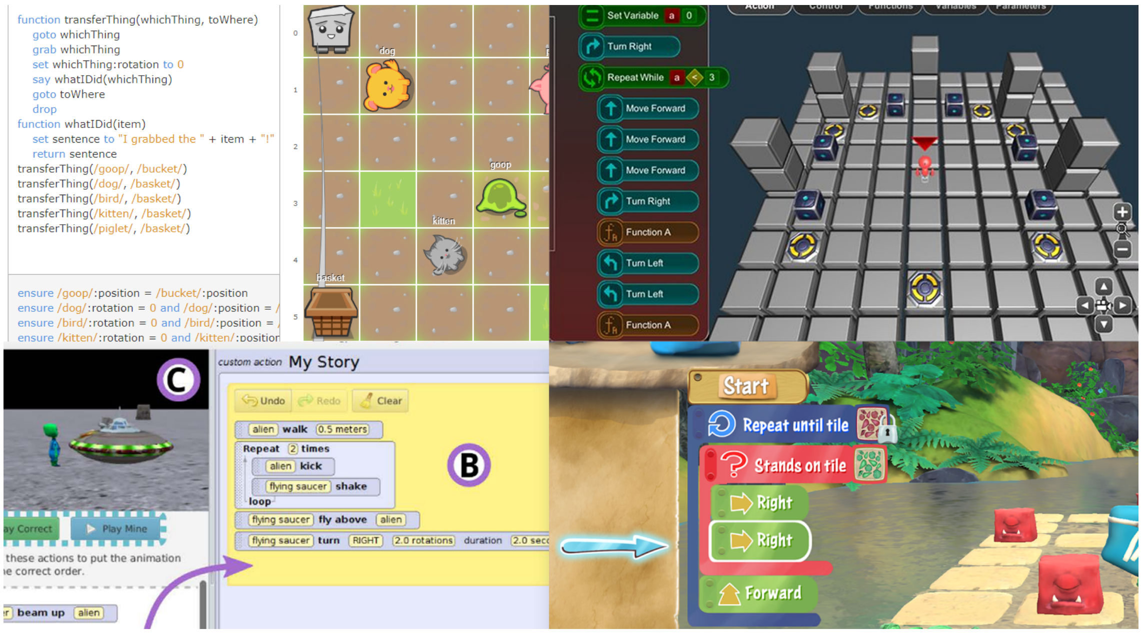
Tools like Dragon Architect attempt to use a hybrid of open-ended exploration and linear puzzles. The player has an open-world in which they can build 3D structures, but are taught how to use certain functionality in a separate level progression (Bauer et al., 2015). Code.org also uses this approach. Tutorial videos are combined with a sequence of progressively more challenging puzzles (Kalelioğlu, 2015) and learners can then use what they have learnt in an open environment. This approach is known as guided discovery, in which discovery learning (similar to constructionism) is paired with in-game guidance. Alfieri et al. (2013) suggest that “unassisted discovery does not benefit learners, whereas feedback, worked examples, scaffolding, and elicited explanations do” (p. 2). These concerns have been echoed by Mayer (2004), who recommends a guided discovery approach with instructional guidance and curricular focus.


Let's build something cool! Click Learn more code to get new tools and abilities, like moving up and down or rotating the camera.

```
turn right
place cube
forward by 1
place cube
forward by 1
place cube
forward by 1
place cube
```

Figure 2.7: Screenshot of Dragon Architect

Other tools use a debugging-first approach (Figure 2.8), where the player has to fix existing broken code to complete tasks (Harms et al., 2015; T. Y. Lee, Mauriello, Ahn, & Bederson, 2014). Box Island has a mechanic where certain instructions are locked in place so the player must design their solution around them. This comes from the notion that debugging is an essential part of learning to program (Fitzgerald et al., 2010) and takes up a sizeable portion of a program’s development time (Du, 2009). Liu, Zhi, Hicks & Barnes (2017) analysed problem-solving behaviours in their programming game, BOTS. They found that debugging requires a deeper understanding than writing new code, meaning that novices should learn better through completing code than by generating new code (Van Merriënboer & De Croock, 1992) if they have some prior experience. This is known as the completion strategy (Paas, 1992), which reduces cognitive load because part of the solution is visible and does not have to be held in working memory.



```
function transferThing(whichThing, toWhere)
    goto whichThing
    grab whichThing
    set whichThing:rotation to 0
    say whatIDid(whichThing)
    goto toWhere
    drop
function whatIDid(item)
    set sentence to "I grabbed the " + item + "!"
    return sentence
transferThing(/goop/, /bucket/)
transferThing(/dog/, /basket/)
transferThing(/bird/, /basket/)
transferThing(/kitten/, /basket/)
transferThing(/piglet/, /basket/)

ensure /goop/:position = /bucket/:position
ensure /dog/:rotation = 0 and /dog/:position = /
ensure /bird/:rotation = 0 and /bird/:position = /
ensure /kitten/:rotation = 0 and /kitten/:position
```

```
Set Variable a 0
Turn Right
Repeat While a < 3
    Move Forward
    Move Forward
    Move Forward
    Turn Right
    Function A
    Turn Left
    Turn Left
    Function A
```

```
custom action My Story
alien walk 0.5 meters
Repeat 2 times
    alien kick
    flying saucer shake
loop
flying saucer fly above alien
flying saucer turn RIGHT 2.0 rotations duration 2.0 sec
```

```
Start
Repeat until tile
    Stands on tile
        Right
        Right
    Forward
```

Figure 2.8: Screenshots of programming tools that follow a debugging-first approach (clockwise from the top-left: Gidget, BOTS, Box Island and Looking Glass)

Design Concept - A Debugging-First Approach
Using a debugging-first approach should help players, because they will be given partially-complete problems that they only need to adjust and extend, instead of starting from scratch.

2.4.7 Physical Devices

Robots, robotics kits and physical block-based languages are also seen as an effective way of teaching programming to novices (e.g. Benitti, 2012; Bers, Flannery, Kazakoff, & Sullivan, 2014). They are often combined with visual programming languages that are either used to program devices or give an on-screen representation of programming blocks. Having a physical artefact makes learning less abstract and more direct, an approach that can be used across all STEM disciplines (Eteokleous, 2019). Bee-bots (Figure 2.9) are widely used in primary schools in England. They are robots that can be programmed to perform a sequence of movements by physically pressing buttons on its back (Mannila et al., 2014). Other examples include KIBO, a robotics kit designed for young children that is programmed using tangible programming blocks and has been shown to be effective in teaching sequencing, logical reasoning and problem-solving skills (Sullivan, Bers, & Mihm, 2017).

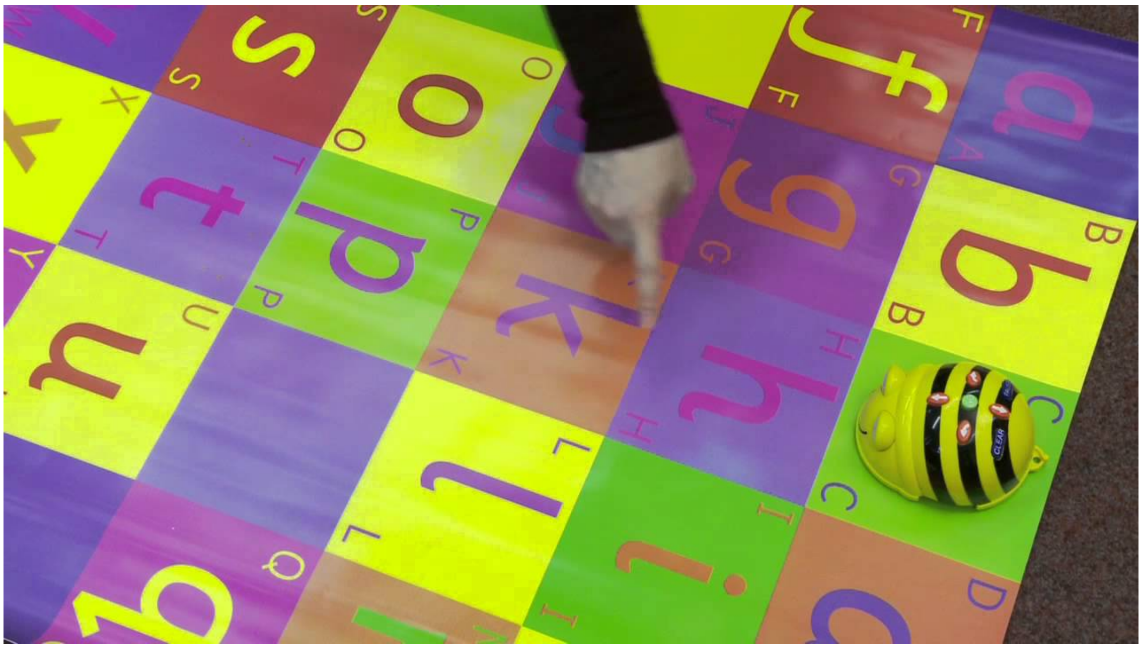


Figure 2.9: Picture of a bee-bot being used with a floor map of letters

2.4.8 Unplugged Activities

Unplugged activities provide ways of exposing students to CS without using computers, through logic games, cards or physical movements (Bell, Alexander, Freeman, & Grimley, 2009). Examples include having children perform a sorting algorithm that results in them lining up in height order or making bracelets coded in binary. Figure 2.10 shows an example from a task sheet from Code.org to develop abstraction skills, creating a generalisation of three sentences. The advantage of unplugged approaches is that schools can deliver CS content without computing equipment, or teachers having technical skills and expertise.

There are indications that unplugged activities improve confidence in primary school children, compared to starting them off with computer-based applications or languages, with learning outcomes staying consistent (Hermans & Aivaloglou, 2017). This finding is supported by Brackmann et al. (2017), who found that unplugged activities improved computational thinking skills compared to a control group.

2.5 Summary

In summary, there is a growing consensus that children should be taught CS in primary education (age 5 to 11). With the aim of giving children an understanding of how technology works, to produce programmers for a growing software engineering sector, to foster logical thinking and problem-solving skills and to give children an awareness of how CS will influence their lives. This has resulted in an increasing number of countries introducing CS into primary education. Children are being taught to create and debug simple programs, using sequence, selection and repetition from around

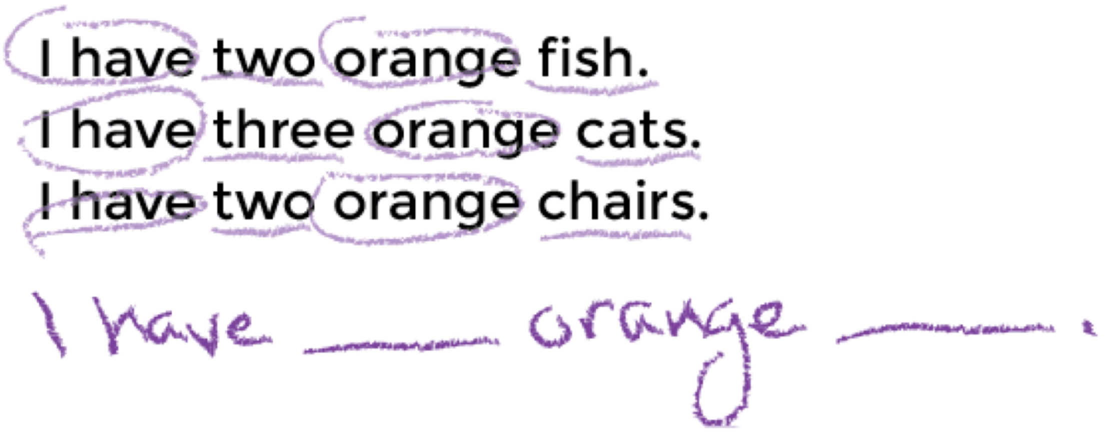


Figure 2.10: Snippet from a Code.org task sheet question on abstraction

5 years old. Curricula also focus on improving logical thinking skills and addressing wider issues of using technology safely and responsibly.

Evidence suggests that primary school children can learn to program at this age and develop abstract reasoning skills if learning content is introduced in a structured and logical way. However, there have been issues with CS teaching, particularly in England and Australia, where curriculum changes were introduced by governments without adequate teacher training in place. The result has been a scramble to produce learning content, teacher training programs and effective and age-appropriate educational programming tools.

There are a wide range of educational programming tools available for primary school children. These fall into four categories; visual programming environments (e.g. Scratch, ScratchJr and Kodu), programming games (e.g. Code.org and Lightbot), physical devices (e.g. Bee-bot) and unplugged approaches. Tools differ in complexity, with those designed for younger children (under age 7) often lacking abstraction and limited to sequencing and algorithms. Visual programming environments and games can differ in learning approach, falling somewhere between 'unassisted discovery' (e.g. Scratch) and linear puzzles that limit freedom and introduce concepts at a steady pace (e.g. Lightbot). The large number of available tools can make it difficult for teachers to know what they should be using in the classroom and how to design learning content around them. From this point on, the thesis will focus on Scratch and Lightbot Jr for the programming interventions. Scratch because it is widely used in primary schools and is readily available. Lightbot Jr because it is age-appropriate for Study 1 (Chapter 4) and meets the aims of teaching the sequencing of programming instructions. Robotics kits were not used for the studies in the thesis as they differ in availability (due to cost) and type between schools.

With the context of CS in primary education in place, the next chapter (Chapter 3) focuses on 'computational thinking'; the idea that the problem-solving and logical thinking skills developed through CS are useful in their own right.

Chapter 3

Computational Thinking

Chapter 2 described several arguments for teaching computer science (CS) in primary education. The 'learner argument', as Passey (2017) called it, is the idea that CS can help develop problem-solving and logical thinking skills. 'Computational thinking' takes this one step further, implying that these skills can be used in a wider context and, along with CS, is a "foundational competency for every child" (Grover, Jackiw, Lundh, & Basu, 2018, p. 1).

Computational thinking (CT) has been used by policymakers as justification for introducing CS into primary education, with learning content focusing on CT as well as CS (Rich et al., 2019). However, it has been criticised by some for its 'decoupling' from the theoretical foundations of CS, along with the lack of evidence for it as a multidisciplinary problem-solving skill (Denning, 2017). CT still lacks a concrete definition (Nardelli, 2019). Yet, current definitions of CT involve working at multiple levels of abstraction, writing algorithms, understanding flow control, recognising patterns and decomposing problems (e.g. Seiter & Foreman, 2013).

This chapter covers the history of CT and explores existing definitions, models and frameworks before giving a working definition for this thesis. It then examines how CT is assessed and measured before exploring some of the criticisms of CT.

3.1 Computational Thinking Origins

The idea that computing's unique methods of thinking can be used as general-purpose 'mental tools' has been around since the conception of computing and CS (Forsythe, 1959). For example, Alan Perlis argued in the 1960s for college students of all disciplines to learn to program, so that the 'theory of computation' could recast "their understanding of a wide variety of topics (such as calculus and economics)" (Guzdial, 2008, p. 25). Denning (2009) suggests that CT was known then as 'algorithmic thinking', that is; "a mental orientation to formulating problems as conversions of some input to an output and looking for algorithms to perform the conversions" (p. 28). Today algorithms and algorithmic thinking make up just a small part of CT definitions.

Papert (1980) was the first to describe these skills as 'computational thinking' while researching how children can develop procedural thinking through computer programming using the Logo programming language (Section 2.4.3).

Wing (2006) sparked a renewed interest in CT, suggesting that "to reading, writing, and arithmetic, we should add computational thinking to every child's analytical ability" (p. 33). This caught the attention of academics and policymakers at a time when technology was becoming cheaper, so it could be viably introduced into schools, and a growing demand for computer-literate professionals in the job market. It is referred to throughout the government reports on computing education in the UK (The Royal Society, 2012) and the US (A. Wilson & Moffat, 2010) mentioned in Chapter 2, which argue for CS inclusion in compulsory education. As such, CT has provided additional justification for including CS into already busy primary school curricula.

CT is seen as the conceptual foundation of CS. Wing suggested that CT is about conceptualising and not programming: "thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction" (2006, p. 34). CS learning content for primary age children often focuses on CT because it is seen as more than just a way of introducing basic programming concepts (Manches & Plowman, 2015). Proponents of CT have suggested that as well as being a key skill for computer scientists, it can benefit problem-solving in other disciplines including mathematics and science. Yet, there is limited evidence for these claims and still no widely agreed-upon definition for CT. The next section examines how academics and policymakers have attempted to define CT.

3.2 Defining Computational Thinking

Wing (2006) deliberately did not give a formal definition in her article, instead describing CT as "solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" (p. 33). Over a decade later, there is still no unanimous agreement on a definition or even a definitive list of the concepts that CT contains (Durak & Saritepeci, 2018). Román-González et al. (2018a) use Aho's (2012) general definition for their development of CT measures: "CT is the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" (p. 834). Despite the ambiguity surrounding CT, it is still playing a key role in defining CS learning content for children (Sentance & Csizmadia, 2017).

The broad consensus is that CT includes all the concepts or thought processes that a computer scientist would typically use to solve computational problems. There have been several efforts to clarify these concepts, in the form of definitions, frameworks and models. This section explores six of these in order to formulate a working definition for the rest of this thesis and also examines some alternative views of CT as a psychological construct. Definitions, frameworks and models are referred to collectively as ‘definitions’ from here on in. Note that some of these definitions use the term ‘K-12’ to describe primary and secondary education (age 5 to 18).

3.2.1 Method

The first three definitions are taken from the most widely-cited papers on CT other than Wing’s articles (2006, 2008). The first two (Barr & Stephenson, 2011; Grover & Pea, 2013) come from early contributors to the CT movement, giving a broad theoretical overview of CT and its application in contexts other than CS. The middle two (Brennan & Resnick, 2012; Seiter & Foreman, 2013) give a more practical perspective. Both are frameworks that use Scratch to show measurable evidence of CT. The final two (Kalelioğlu, Gülbahar, & Kukul, 2016; Shute, Sun, & Asbell-Clarke, 2017) are more recent and both come from literature reviews on the topic, giving a more up-to-date view of CT.

3.2.2 Definitions, Frameworks & Models

Barr & Stephenson (2011)

Barr & Stephenson (2011) describe the CT definition that came from a 2009 joint project between the Computer Science Teachers Association (CSTA) and the International Society for Technology in Education (ISTE). The project brought together 26 academics and educators to produce an operational definition for CT and to define the steps needed to apply this in K-12 education. They defined CT as “an approach to solving problems in a way that can be implemented with a computer” and suggested that it can be applied in “every other type of reasoning” (p. 115).

The project produced a model of the core CT concepts and examples of where they might be embedded in other disciplines, including CS, mathematics, science, social studies and language arts. The CS examples have been included with each concept:

Data collection - Find a data source for the problem area.

Data analysis - Write a program to do basic statistical calculations on a set of data.

Data representation and analysis - Use data structures such as an array, linked list, stack, queue, graph, hash table, etc.

Abstraction - Use procedures to encapsulate a set of often-repeated commands that perform a function.

Analysis and model validation - Validate a random number generator.

Automation - *Example not given.*

Testing and verification - Debug a program: write unit tests; formal program verification.

Algorithms and procedures - Study classic algorithms; implement an algorithm for a problem area.

Problem decomposition - Define objects and methods; define main and functions.

Control structures - Use conditionals, loops, recursion, etc.

Parallelisation - Threading, pipelining, dividing up data or task in a such a way to be processed in parallel.

Simulation - Algorithm animation, parameter sweeping.

Design Concept - Teaching Abstraction using Procedures
Having programming tasks that teach players to use procedures will give them a concrete way of using abstraction.

They then listed ways in which students can demonstrate CT:

- Design solutions to problems using abstraction, automation, creating algorithms, data collection and analysis.
- Implement designs (programming as appropriate).
- Test and debug.
- Model, run simulations, do systems analysis.
- Reflect on practice and communicating.
- Use the vocabulary.
- Recognise abstractions and move between levels of abstractions.
- Innovation, exploration, and creativity across disciplines.
- Group problem solving.
- Employ diverse learning strategies.

They also describe dispositions and pre-dispositions to capture the “areas of values, motivations, feelings, stereotypes and attitudes” (p. 118) applicable to CT:

- Confidence in dealing with complexity.

- Persistence in working with difficult problems.
- The ability to handle ambiguity.
- The ability to deal with open-ended problems.
- Setting aside differences to work with others to achieve a common goal or solution.
- Knowing one's strengths and weaknesses when working with others.

Grover & Pea (2013)

Grover & Pea's (2013) literature review was one of the first on CT. They aimed to frame the current state of discourse on CT in K-12 education using Wing's 2006 article as a springboard. They state that abstraction is what distinguishes CT from other types of thinking, and go on to state the elements that are "now widely accepted as comprising CT and form the basis of curricula that aim to support its learning as well as assess its development" (p. 39):

- Abstractions and pattern generalisations (including models and simulations).
- Systematic processing of information.
- Symbol systems and representations.
- Algorithmic notions of flow control.
- Structured problem decomposition (modularising).
- Iterative, recursive, and parallel thinking.
- Conditional logic.
- Efficiency and performance constraints.
- Debugging and systematic error detection.

Brennan & Resnick (2012)

Brennan & Resnick (2012) were interested in the way that design-based learning tasks can support CT in young people. Particularly focusing on strategies for assessment. They developed their framework by watching and interviewing Scratch (Section 2.4.4) users age 8 to 16 over several years. The authors see CT as "a device for conceptualising the learning and development that takes place using Scratch" (p. 2), although their framework has been applied in more general CT analyses (e.g. Da Cruz Alves, Gresse Von Wangenheim, & Hauck, 2019; Falloon, 2016). The framework has three dimensions: computational concepts (employed when programming), computational practices (developed when programming) and computational perspectives (formed about the world and the programmer themselves). They go on to discuss an approach to assessing these dimensions using project portfolio analysis, artifact-based interviews and design scenarios.

The computational concepts they list are common in most programming languages (and map to Scratch programming blocks) and are useful in a range of programming and non-programming contexts:

Sequences - Activity or task expressed in a series of steps that can be executed by a computer.

Loops - Running sequences multiple times.

Events - Triggering things to happen when something else happens.

Parallelism - Sequences of instructions happening at the same time.

Conditionals - Making decisions based on certain conditions.

Operators - Mathematical, logical and string expressions.

Data - Storing, retrieving and updating values.

Computational practices describe how the learner is thinking and learning:

Being incremental and iterative - Approaching a problem in steps.

Testing and debugging - Developing strategies for dealing with problems.

Reusing and remixing - Building on the work of others.

Abstracting and modularising - Dealing with a problem by breaking it down into smaller parts.

The final part of the framework is computational perspectives: changes in the learners understanding of themselves and the world around them:

Expressing - Seeing computation as a medium for creative expression.

Connecting - Using the ideas and influence of others to produce better content.

Questioning - Asking questions of why and how things work.

Seiter & Foreman (2013)

Seiter & Foreman's (2013) Progression of Early Computational Thinking (PECT) model assumes that every student has a latent proficiency in CT that manifests itself in their ability to complete specific tasks. It maps measurable evidence (in Scratch) onto more abstract coding design patterns, which are then mapped onto CT concepts. PECT is designed to measure CT amongst students at primary school level (age 5 to 11).

There are three levels of assessment, which are ordered in decreasing levels of abstraction (most abstract first):

1. CT concepts
2. Design patterns
3. Evidence variables (explicit programming constructs)

The CT concepts are a subset of those proposed by Barr & Stephenson (2011), with 'process skills' such as testing and verification removed because they are difficult to collect evidence for:

- Procedures and algorithms
- Problem decomposition
- Parallelisation and synchronisation
- Abstraction
- Data representation

The model then lists design patterns (common coding patterns) that are often used in Scratch:

- Animate looks
- Animate motion
- Conversate
- Collide
- Maintain score
- User interaction

Students demonstrate CT by choosing the correct design pattern for a problem or context and then implementing it successfully. They are assessed on a quantitative scale for each design pattern:

1. **Basic** - Functional understanding.

2. **Developing** - Advanced but not complete understanding.
3. **Proficient** - Complete understanding.

Evidence variables are used to measure the concrete computational aspects of a student's work and contribute to the scores for each design pattern. A score is given for each category using a proficiency rating similar to the design patterns. The variables are roughly organised into Scratch block categories:

- Looks
- Sound
- Motion
- Sequence & looping
- Boolean expressions
- Operators
- Conditional
- User interface event
- Parallelisation
- Initialise location
- Initialise looks

Kalelioğlu, Gulbahar & Kukul (2016)

Kalelioğlu, Gulbahar & Kukul (2016) state that "CT literature is at an early stage of maturity, and is far from either explaining what CT is, or how to teach and assess this skill" (p. 583). They reviewed existing research on the topic, finding that most provided activities (plugged or unplugged) to promote CT in K-12 education, without much empirical support. From their review, they produced a framework that describes CT as a problem-solving process (Table 3.1) and can be used "in either a computerised or unplugged approach" (p. 592).

Shute, Sun & Asbell-Clarke (2017)

Shute, Sun & Asbell-Clarke's (2017) literature review of CT is the most recent at the time of writing. They describe the literature as showing "a diversity in definitions, interventions, assessments, and models" (p. 142) and go on to define CT as "the conceptual foundation required to solve problems effectively and efficiently (i.e., algorithmically, with or without the assistance of computers) with solutions that are reusable

Table 3.1: Computational thinking as a problem-solving process (from left to right) (Kalelioğlu, Gülbahar, & Kukul, 2016)

Identify the problem	Gathering, representing and analysing data	Generate, select and plan solutions	Implement solutions	Assessing solutions and continue for improvement
Abstraction	Data collection	Mathematical reasoning	Automation	Testing
Decomposition	Data analysis	Building algorithms and procedures	Modelling and simulations	Debugging
	Pattern recognition	Parallelisation		Generalisation
	Conceptualising			
	Data representation			

in different contexts" (p. 151). They describe CT as a way of thinking and acting that can be exhibited through particular skills including engineering and mathematics. Table 3.2 shows their summary of the facets included in their definition.

3.2.3 Differences Between Definitions

There are several noticeable differences between the definitions analysed in Section 3.2.2. The scope of CT ranges from conceptualising learning and development in a specific language (e.g. Scratch) (Brennan & Resnick, 2012; Seiter & Foreman, 2013), where CT is tightly coupled to the programming implementation, to an all-encompassing term that includes engineering, mathematics and design thinking (Shute et al., 2017). CT is described as a problem-solving process (Kalelioğlu et al., 2016), a latent cognitive ability (Seiter & Foreman, 2013) and as a broad term for a range of skills and problem-solving approaches (Grover & Pea, 2013). This lack of clarity is one of the criticisms of CT that is expanded upon in Section 3.4, making it difficult to produce measurable learning outcomes that are not embedded in programming tasks.

Table 3.3 shows the main CT concepts in each definition. The next section draws out the common concepts from this table and gives a working definition of CT for this thesis.

Table 3.2: Computational thinking facets and their definitions (Shute, Sun, & Asbell-Clarke, 2017)

Facet	Definition
Decomposition	Dissect a complex problem/system into manageable parts.
Abstraction	Extract the essence of a (complex) system. Includes: • Data collection and analysis (collect the most relevant and important information) • Pattern recognition (identify patterns/rules underlying the data/information structure) • Modelling (build models or simulations to represent how a system operates)
Algorithms	Design logical or ordered instructions for rendering a solution to a problem, these can be carried out by a human or a computer. Includes: • Algorithm design (create a series of ordered steps to solve a problem) • Parallelism (carry out a certain number of steps at the same time) • Efficiency (design the fewest number of steps to solve a problem, removing unnecessary or redundant steps) • Automation (automate the execution of the procedure when required to solve similar problems)
Debugging	Detect, identify and fix errors.
Iteration	Repeat design processes to refine solutions.
Generalisation	Transfer CT skills to a wide range of situations/domains to solve problems effectively and efficiently.

3.2.4 A Working Definition

The concepts included in over half the definitions in Table 3.3 have been used to form a working definition of CT. These include abstraction and generalisation, algorithms and procedures, data collection, analysis and representation, parallelism, decomposition, debugging, testing and analysis and control structures. Table 3.4 shows the CT concepts, a description of what they involve and how many of the analysed definitions they were included in. Using these concepts, this thesis will take a temperate approach to CT, defining it as “the thought processes involved in modelling and solving computational problems.” This definition implies that CT concepts (or thought processes) are not language-specific (e.g. Scratch) but can be applied to all computational problems.

Table 3.3: Computational thinking concepts included in the definitions, models and frameworks from Section 3.2.2

Barr & Stephenson (2011)	Brennan & Resnick (2012)	Grover & Pea (2013)	Seiter & Foreman (2013)	Kalelioğlu, Gulbahar & Kukul (2016)	Shute, Sun & Asbell-Clarke (2017)
Abstraction	Abstracting and modularising	Abstraction and pattern generalisation	Abstraction	Abstraction	Abstraction
Algorithms and procedures	Sequences	Algorithmic notions of flow control	Procedures and algorithms	Algorithms and procedures	Algorithms
Data collection, analysis and representation	Data	Symbol systems and representations	Data representation	Data collection, analysis and representation	Data collection and analysis
Problem decomposition		Structured problem decomposition	Decomposition	Decomposition	Decomposition
Parallelisation	Parallelism	Iterative, recursive and parallel thinking	Parallelisation and synchronisation	Parallelisation	Parallelism
Testing and verification	Testing and debugging	Debugging and systematic error detection		Testing and debugging	Debugging
Control structures	Conditionals and loops	Conditional logic		Mathematical reasoning	
Automation				Automation	Automation
				Generalisation	Generalisation
Simulation				Modelling and simulations	Modelling
	Events				
		Efficiency and performance constraints			Efficiency
		Systematic processing			
				Conceptualising	
					Iteration

Table 3.4: Concepts included in the working definition of computational thinking

Concept	Meaning	Included In
Abstraction and generalisation	Removing the detail from a problem and formulating solutions in generic terms.	6/6
Algorithms and procedures	Using sequences of steps and rules to solve a problem.	6/6
Data collection, analysis and representation	Using and analysing data to help solve a problem.	6/6
Parallelism	Having more than one thing happening at once.	6/6
Decomposition	Breaking a problem down into parts.	5/6
Debugging, testing and analysis	Identifying, removing and fixing errors.	5/6
Control structures (and mathematical reasoning)	Using conditional statements and loops.	4/6

3.2.5 A View from Cognitive Psychology

An alternative view of CT is as an emerging psychological construct. Some researchers have tried to break it down into cognitive processes that can be tested using a battery of existing assessments. Ambrósio, Xavier & Georges (2014) suggest that CT is related to three abilities-factors from the Cattell-Horn-Carroll (CHC) model of intelligence (McGrew, 2009): fluid reasoning, visual processing and short-term memory. Román-González, Pérez-González, & Jiménez-Fernández (2016) built on this, finding that scores from their CT test correlated strongly with verbal, spatial and reasoning factors from the Primary Mental Abilities (PMA) battery (Thurstone, 1938) and also with scores from the RP30 problem-solving test, which requires reasoning, spatial ability and working memory.

Executive Functions

There are also indications that CT is related to executive functions: higher-order cognitive functions including holding and manipulating information in working memory and attention shifting (cognitive flexibility). Executive function is a predictor of academic success in general (Cragg & Gilmore, 2014). Robertson (2019) found correlations between assessments of programming and debugging in Scratch and scores from CANTAB and BRIEF 2 assessments of executive functions in children age 11. The Scratch assessments were a creative programming task measured for CT using Dr. Scratch (Section 3.3.1) and a set of seven debugging tasks where the participant had to locate and fix an error. A link between executive functions and CT would explain the difficulties that children have with more abstract programming concepts, particularly those that require working memory and attention shifting. Working memory appears to develop gradually between age 4 and 7 (Luciana & Nelson, 1998) and then continues to improve up to age 14 (De Luca et al., 2003).

Section 2.4.6 discusses programming tools that use a 'debugging-first' approach. There is an argument that debugging code requires a deeper understanding than writing new code because the programmer must be able to understand the code fully to locate the error. Debugging requires the programmer to develop a plan of detecting, fixing and testing that places a high demand on working memory and requires them to shift their attention between different representations of the code. These tasks, therefore, require better executive functions. Grover et al. (2015) suggest that unstructured programming tasks by themselves do not improve CT, which means that debugging tasks with minimal guidance should produce better learning outcomes than 'discovery' learning.

Design Concept - Debugging-First with Guidance
The player should be given guidance to help them complete structured debugging-first tasks.

3.3 Measuring Computational Thinking

Whilst there have been some attempts to establish the cognitive underpinnings of CT (Román-González et al., 2016) (Section 3.2.5), there is currently a lack of CT measures that have been shown to be reliable and valid due to the immaturity of the field (Allsop, 2018). Existing CT measures can be categorised as either formative, skill-transfer or summative. Table 3.5 shows the available measures, giving a description of each, associated programming tools and a summary of studies for validity and reliability. The section then gives an overview of each category and describes an example measure for each.

Table 3.5: Existing computational thinking measures

Measure	Author(s)	Description	Tool	Validity/Reliability
		Formative		
Fairy	Werner, Denner & Campe (2012)	Tasks in an Alice program	Alice	None
Three-Dimensional Integrated Assessment (TDIA)	Zhong et al. (2016)	Framework of tasks	Any (Alice used by authors)	None
Computational Thinking Practices Design Patterns	Bienkowski et al. (2015)	Design patterns	Any (Scratch, AgentSheets & Alice used so far)	None
Progression of Early Computational Thinking (PECT) Model	Seiter & Foreman (2013)	Model for assessing Scratch projects	Scratch	None
Dr. Scratch	Moreno-León & Robles (2015)	Analysis tool	Scratch	Ecological validity, convergent validity and discriminate validity (Robles et al., 2018)

Continued on next page

Continued from previous page

Measure	Author(s)	Description	Tool	Validity/Reliability
Portfolio analysis, interviews and design scenarios	Brennan & Resnick (2012)	Analysis framework, interviews and design scenarios	Scratch	None
		Skill-Transfer		
Bebras Computing Challenge	Dagiene & Futschek (2008)	Test (written questions - multiple choice)		Used in several CT studies (e.g. Boylan, Demack, Wolstenholme, Reidy, & Reaney-Wood, 2018; Straw, Bamford, & Styles, 2017)
Computational Thinking Pattern Quiz	Basawapatna et al. (2011)	Test (video questions)		None
		Summative		
Computational Thinking test (CTt)	Román-González et al. (2018b)	Test (visual programming - multiple choice)		Content validity and criterion validity (Román-González et al., 2016)
Test for Measuring Basic Programming Abilities	Mühling, Ruf & Hubwieser (2015)	Test (visual and text-based programming)		None

Continued on next page

Continued from previous page

Measure	Author(s)	Description	Tool	Validity/Reliability
Commutative Assessment Test	Weintrop & Wilensky (2015)	Test (visual and text-based programming		None
SOLO taxonomy	Biggs & Collis (2014)	Project analysis using categories	Any (used with Scratch (Seiter, 2015))	Not for CT

3.3.1 Formative Measures

Formative measures provide feedback for learners to improve their CT skills. These are often project-based, working on the assumption that learners use CT in the completion of design tasks. Learners create projects using a programming tool (e.g. Scratch). Their projects are then analysed for their use of CT, either manually or using automated software. These scores can then be used to address areas of weakness in projects and can motivate the learner to learn new functionality. Table 3.5 includes several examples of formative measures, predominantly for Alice and Scratch.

Dr. Scratch will be discussed in more detail as there is some evidence for its reliability and validity as a measure of CT in Scratch projects.

Table 3.6: Dr. Scratch scoring system

CT Concept	Basic (1 point)	Developing (2)	Proficiency (3)
Logical thinking	If	If else	Logic operations
Data representation	Modifiers of sprite properties	Variables	Lists
User interactivity	Green flag	Keyboard, mouse, ask and wait	Webcam, input sound
Flow control	Sequence of blocks	Repeat, forever	Repeat until
Abstraction and problem decomposition	More than one script and more than one sprite	Procedures (definition of own blocks)	Use of clones
Parallelism	Two scripts on green flag	Two scripts on key pressed or the same sprite clicked	Two scripts on when I receive message, or video or input audio, or when backdrop changes to
Synchronisation	Wait	Message broadcast, stop all, stop program	Wait until, when backdrop changes to, broadcast and wait

Dr. Scratch

Dr. Scratch (Moreno-León & Robles, 2015) is an automated tool that gives Scratch projects a score out of 21 across seven CT concepts based on the blocks used (Table 3.6). The CT concepts used are similar to those in the working definition of this thesis. The Dr. Scratch authors have validated their tool for its ecological validity (usefulness for learners), convergent validity (comparable to expert judgements of projects and software engineering complexity metrics) and discriminate validity (comparable scores from different types of Scratch projects) (Robles et al., 2018). Dr. Scratch has been used in several recent studies of CT with primary and secondary school children (e.g. Förster, Förster, & Loewe, 2018; Lawanto, Close, Ames, & Brasiel, 2017).

However, Dr. Scratch gives no guarantee that the user understands the blocks that they are using to get points in a certain category. Blocks can be used incorrectly in a program and still gain the user points for that CT concept. Furthermore, the 'best' way to complete a task may not involve any blocks that get the user a high Dr. Scratch score, which should be considered when measuring projects that have been created to a specification. Yet, it does fit with the constructionist, open design of Scratch where the user is encouraged to experiment with and explore functionality.

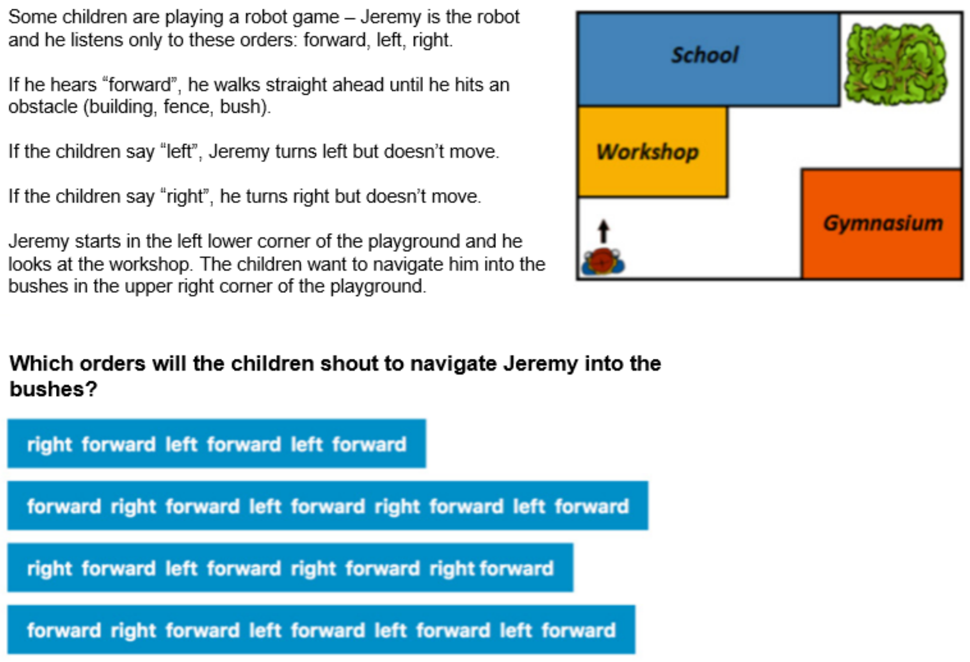
3.3.2 Skill-Transfer Measures

Skill-transfer measures are aimed at assessing a learner's ability to apply CT in different contexts. These are either assessments, questionnaires or surveys and are often self-reporting or text-heavy and reliant on comprehension. Examples include the Bebras Computing Challenge (Dagiene & Futschek, 2008) and the Computational Thinking Pattern Quiz (Basawapatna et al., 2011). Bebras has been used recently as a measure of CT in two large educational studies (Boylan et al., 2018; Straw et al., 2017), so will be discussed in more detail.

Bebras Computing Challenge

The Bebras computing challenge is an international contest that aims to introduce informatics and CT to children age 6 to 18 (Dagiene & Stupuriene, 2016). It had over 2 million participants in 2018 (Bebras, 2018). Contestants are given between 15 and 21 questions of different difficulty levels to solve in 45 minutes (an example is shown in Figure 3.1). The questions do not require any pre-requisite knowledge. Hubweiser & Mühling (2015) suggest some suitable sets of these questions could be assembled that would measure to CT standards outlined by the CSTA (Barr & Stephenson, 2011) and that Bebras could be the basis of future PISA (Programme for International Student Assessment) assessments for CS (Hubwieser & Mühling, 2014).

Straw, Bamford & Styles (2017) used Bebras in a large randomised controlled trial to measure the impact of attending code clubs on CT skills in children age 9 and 10, finding that "attending Code Club for a year did not have an impact on pupils' computational thinking over and above changes that would have occurred anyway" (p. 5). However, Boylan et al. (2018) found that scores improved on a subset of Bebras tasks after a year of using Scratch with children of the same age, supporting Hubweiser & Mühling's (2015) suggestion that suitable sets of Bebras questions can be used to measure CT.


Some children are playing a robot game – Jeremy is the robot and he listens only to these orders: forward, left, right.

If he hears "forward", he walks straight ahead until he hits an obstacle (building, fence, bush).

If the children say "left", Jeremy turns left but doesn't move.

If the children say "right", he turns right but doesn't move.

Jeremy starts in the left lower corner of the playground and he looks at the workshop. The children want to navigate him into the bushes in the upper right corner of the playground.

Which orders will the children shout to navigate Jeremy into the bushes?

right forward left forward left forward

forward right forward left forward right forward left forward

right forward left forward right forward right forward

forward right forward left forward left forward left forward

Figure 3.1: Question from the Bebras computing challenge

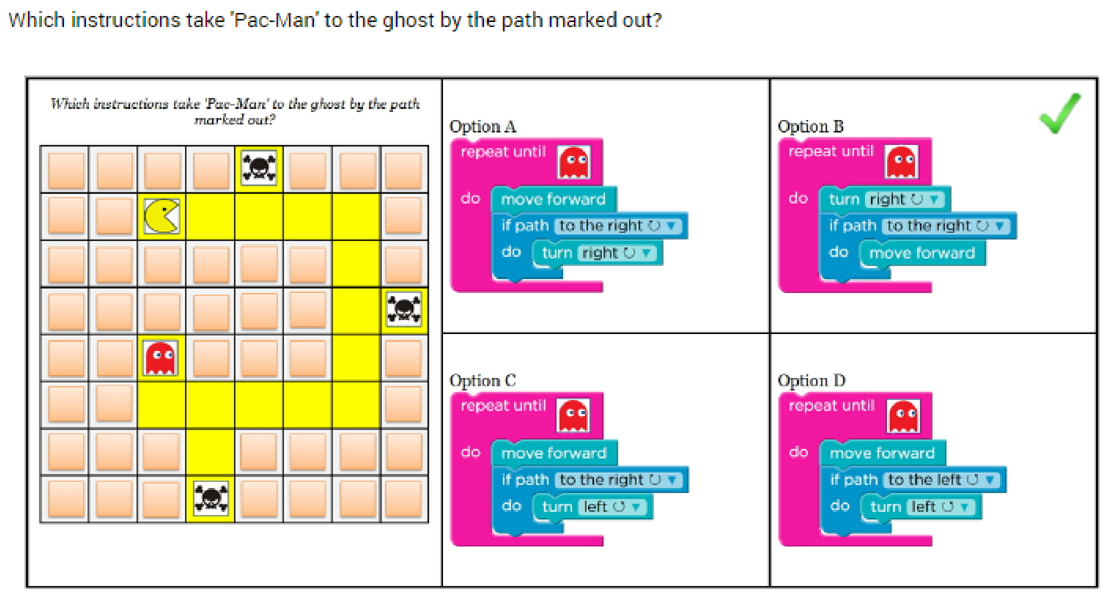
3.3.3 Summative Measures

Some summative measures are designed to test a learner's CT aptitude. These include the Computational Thinking test (Román-González et al., 2018b), the Test for Measuring Basic Programming Abilities (Mühling et al., 2015) and the Commutative Assessment Test (Weintrop & Wilensky, 2015). Others measure a learner's understanding of computational concepts using a programming tool (e.g. Scratch). These can use adaptations of Bloom's taxonomy (Anderson et al., 2009), like the SOLO taxonomy (Biggs & Collis, 2014) to assess higher-order thinking used in projects (e.g. Seiter, 2015). Some of the formative measures described in Section 3.3.1, such as Dr. Scratch, can also be used summatively.

The Computational Thinking test is discussed in more detail as it has been used in several studies of CT and programming (e.g. Pérez-Marín, Hijón-Neira, Bacelo, & Pizarro, 2018).

Computational Thinking Test

The Computational Thinking test (CTt) aims to measure CT as "the ability to formulate and solve problems by relying on the fundamental concepts of computing, and using logic-syntax of programming languages: basic sequences, loops, iteration, conditionals, functions and variables" (Román-González et al., 2016, p. 4). It contains 28 multiple-choice questions (Román-González, 2016) and has been used with children age 10 to 14 (e.g. Brackmann et al., 2017). Each question has four options and one correct answer. Questions use either visual arrows or visual blocks that are common in educational programming tools (Section 2.4). The authors have conducted studies of the predictive validity of the CTt with regards to academic performance and a Code.org course, suggesting that CTt can be used to detect ‘computationally talented’ students in middle school (Román-González et al., 2018a).


Which instructions take 'Pac-Man' to the ghost by the path marked out?

Which instructions take 'Pac-Man' to the ghost by the path marked out?

Figure 3.2: Question from the Computational Thinking test

The CTt is closely tied with block-based visual programming (an example question is shown in Figure 3.2). However, Román-González, Moreno-León, & Robles (2017) suggest that the CTt can be combined with Dr. Scratch (formative) and Bebras (skill-transfer) as an overall measure of CT. They found significant correlations between the three, suggesting that they are partially convergent.

3.4 Criticisms of Computational Thinking

There are several concerns with CT. Whilst, in theory, concepts like abstraction, decomposition and debugging can be useful in a wider context, there are unanswered questions on transfer, learning content, assessments, views of it as a universal skill and teacher expertise. These issues are discussed in the context of CS in primary education (Chapter 2).

3.4.1 Does Computational Thinking Transfer?

CT is receiving widespread attention and is seen by some as a necessary problem-solving tool for every child (Grover et al., 2018). But as of yet, there are no data to support claims that CT can help solve non-computational problems (Denning, 2017). Historically, a number of studies in the 1980s found no evidence that problem-solving skills developed through programming in Logo are transferable to non-computational domains (e.g. Clements & Gullo, 1984; Pea & Kurland, 1984). There can also be difficulties with transfer from one programming language to another (Shrestha, Barik, & Parnin, 2018), two skills that obviously require CT. Scherer (2016) argues that despite computer programming and other skills sharing cognitive and meta-cognitive processes, educational research is lagging in providing evidence of transfer. The immaturity of the field is a factor, Kalelioğlu, Gulbahar & Kukul (2016) found that most CT publications from the last ten years were lacking theoretically, conceptually or in terms of in-depth research and empirical evidence. Additionally, it is difficult to show that CT skills benefit learners in other disciplines like mathematics and science without valid and reliable assessments.

3.4.2 Is Computational Thinking Separate From Computer Science?

Armoni (2016) argues that "CT curricula are mostly based on programming, and seldom, if at all, explicitly specify high-level CT strategies" (p. 26). Several of the definitions in Section 3.2.2 are tightly coupled with programming, and in particular, Scratch. Seiter & Foreman (2013) suggest that CT is a latent proficiency and can be measured only through application, of which the most obvious method is programming. Unplugged approaches (Section 2.4.8) are seen as a way of teaching CT and are widely available as part of resources such as Code.org and Barefoot (Barefoot Computing, 2019). Brackmann et al. (2017) found evidence that unplugged approaches can work to teach CT using the visual programming-centred CTt as their measure. It is, therefore, not clear whether their unplugged approach improved CT in terms of transfer to other domains, or just its application in CS.

Denning, Tedre & Yongpradit (2017) suggest that thinking of any step-by-step procedure as an algorithm (e.g. making a jam sandwich) is a mistake. The steps in an algorithm should be machine-executable, and a 'step' in the human sense of the word, as an isolated action of a person, implies that computers can do a lot more than they actually can. This is an important distinction because unplugged curricula often confuse the two. For example, Barefoot (2019), who provide CT resources for primary and secondary education, have an 'algorithm' activity for sharing sweets, where one step in the algorithm may be 'snatch as many as you can'. This is obviously a human step and not a machine-executable one. Misconceptions like this can lead to exaggerations of what CT is capable of. Shute, Sun & Asbell-Clarke (2017) describe the differences between CT and other types of thinking, suggesting that CT is "an umbrella term containing design thinking and engineering (i.e., efficient solution design), systems thinking (i.e., system understanding and modelling), and mathematical thinking as applied to solving various problems" (p. 146). However, it is difficult to find experimental evidence to justify this view.

Armoni (2016) suggests that CT should not be separated from CS and that attempting to operationalise CT as a new discipline causes more harm than good. CT should instead be seen as an explanation of the benefits of CS. Nardelli (2019) supports this view, suggesting that CT should not be separated from CS, just as mathematical thinking or linguistic thinking are not separated from mathematics or linguistics.

3.4.3 Assessing Computational Thinking?

CS and CT are often not formally assessed in primary education in the same way as mathematics and literacy, even in countries where CS has its own subject. When it is assessed, there is no standard approach and assessments can take many different forms, including the formative, skill-transfer and summative measures discussed in Section 3.3. Kallia (2017) expands on this, listing other types of CT assessment that can be used: self-assessment, peer-assessment, Bloom's & SOLO taxonomy, isomorphic questions, parametrised questions, rubrics, automated tools for programming, assessment tools for CT, concept maps, code comprehension, debugging tasks and multiple-choice questions and quizzes.

Of the measures in Section 3.3, some are project-based and tightly coupled with programming. These formative measures take different forms that are often dependent on the programming tool (e.g. Scratch). Others focus on CT aptitude (summative), through a series of multiple-choice questions, or CT application in different contexts using written comprehension tasks (skill-transfer). The wide-range of assessments leads to inconsistencies with what CT means, an issue made worse by the lack of a formal definition. There is no guarantee, without experimental evidence, that because learners improve on one measure that they will improve on another.

The removal of 'process skills' such as testing and verification from Seiter & Foreman's (2013) PECT model highlights one of the problems with measuring CT; if CT is a problem-solving process, then it is difficult to accurately measure the process that someone follows before arriving at a solution. Only the solutions themselves can be quantified. Brennan & Resnick (2012) state that accurately measuring CT requires a combination of measures because just looking at a student project does not demonstrate all of their computational competencies. Moreno-León, Román-González & Robles (2018) suggest a combination of CTt, Bebras and Dr. Scratch to give an overall measure of CT. However, conducting a combination of three measures is time-consuming, which can be an issue in the classroom. Identifying the psychological constructs that underpin CT may help measure the problem solving that goes into creating solutions, for example, fluid reasoning, visual processing and short-term memory (Román-González et al., 2016) or executive functions (Robertson, 2019). Yet, this research has only shown correlations between these skills and CT, which could be due to these measures being influenced by other factors.

These issues affect curriculum design because it is difficult to establish what should be taught. The lack of valid and reliable CT measures also makes it difficult to properly test the efficacy of educational programming tools.

3.4.4 Is Computational Thinking a Universal Skill?

Wing suggested that CT is a "universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use" (Wing, 2006, p. 33), a view echoed more recently by Grover et al. (2018). However, there is currently little evidence to support claims that CT improves general cognitive skills and can help people perform everyday tasks (Guzdial, 2015). This means that it is unclear whether everyone will benefit from being able to think computationally (Webb et al., 2017). Yet, Chapter 2 suggested that there are benefits to all children having a basic understanding of CS in an increasingly technology-based society, even if learning outcomes are unclear.

The indication that CT is related to executive functions (Robertson, 2019) suggests that CT materials should be designed around the cognitive limitations of younger learners. CT gives us a breakdown of problem-solving skills that are used in CS, which may allow educators to focus on developing individual competencies in younger children who struggle with more abstract concepts (e.g. variables and functions). For example, focusing on 'sequencing' or simple algorithms and seeing if this can transfer to more 'universal' skills used in mathematics and science. Kazakoff, Sullivan & Bers (2012, 2014; 2013) pursued this line of enquiry, finding that a robotics programming tool improved story sequencing in children aged between 4 and 6. Chapter 4 describes a study that attempts to replicate these results using the Lightbot Jr programming game.

3.4.5 Is Primary School Teaching of Computational Thinking Effective?

The lack of teacher expertise in primary-level CS education is discussed in Section 2.3. In summary, most teachers lack the CS knowledge needed to support learning. This is because, in some countries, CS and CT have been introduced into schools by policymakers without adequate training resources, learning materials and infrastructure. As it is not yet an assessed part of the curriculum, like mathematics or literacy, there is little motivation within schools to dedicate time and money towards it (Webb et al., 2017).

Several studies have been done on teacher attitudes towards CS and CT in primary education. Sentance & Csizmadia (2017) analysed 339 teacher responses to a survey on the strategies and challenges of teaching CS in compulsory education in the

UK. They found that the main concern was the teachers' own subject knowledge. Respondents reported that they often spent hours of their own time trying to upskill in the subject. These concerns were echoed in a similar survey by Rich et al. (2019), who analysed 310 teacher responses mainly from the US and UK and found that teachers' greatest concern was their own ability to learn computing/coding. US teachers were also unsure how and where in the curriculum they should be teaching CS. The authors found that a wide range of programming tools were used and that CS/CT was often integrated into mathematics or science. Yadav et al. (2016) investigated CS teacher perspectives in the US, finding that teachers struggled with developing adequate knowledge, not having sufficient IT software in schools and feeling isolated because of having to train themselves and find appropriate resources. Teachers felt unable to effectively support student learning unless they sought training and found resources in their own time.

In a separate piece, Yadav, Stephenson and Hong (2017, p. 60) give five recommendations for improving CT education:

> **Curriculum** - Develop a pre-service teacher education curriculum to prepare teachers to embed CT in their classrooms.
>
> **Core ideas** - Introduce pre-service teachers to core ideas of CT by redesigning educational technology courses.
>
> **Methods courses** - Use elementary and secondary methods courses to develop pre-service teachers' understanding of CT in the context of the discipline.
>
> **Collaboration** - Computer science educators and teacher educators collaborate on developing CT curricula that goes beyond programming.
>
> **Teacher education** - Use existing resources and curriculum standards to assimilate CT into pre-service teacher education.

However, these recommendations bring us back to the issues discussed in this section. Does CT transfer to other disciplines? How and when should it be assessed? And is it a universal skill that every child should develop? Without definitive answers to these questions, it is difficult to justify centring CS education on CT. This is a key factor behind Armoni (2016) and Nardelli's (2019) suggestions that CT should be viewed as an explanation of the benefits of CS and not as a new discipline, where CT ability will improve as the learner becomes more proficient in CS. This, combined with the lack of teacher expertise at primary level, highlights the importance of well-designed and thoroughly-tested educational programming tools.

3.5 Summary

In summary, CT encompasses the problem-solving concepts used by computer scientists to solve computational problems. For the purpose of this thesis, CT is defined as “the thought processes involved in modelling and solving computational problems” and includes abstraction and generalisation, algorithms and procedures, data collection, analysis and representation, decomposition, parallelism, debugging, testing and analysis and using control structures. As of yet, there is limited evidence of transfer to non-computational domains, despite it being clear that CT is used in CS.

The concepts included in CT, and its scope, are still unclear. Some suggest that it can be used to conceptualise learning and development in a specific language (e.g. Scratch), whereas others suggest it is an all-encompassing conceptual foundation that includes engineering, mathematics and design thinking (Shute et al., 2017). There are also views of it as an emerging psychological construct that correlates with fluid reasoning, visual processing and short-term memory (Moreno-León et al., 2018) and executive functions (Robertson, 2019). There are several existing CT measures, but they lack evidence of validity and reliability at the time of writing. These include formative project-based measures like Dr. Scratch, skill-transfer measures like Bebras, and summative measures like the Computational Thinking test.

In terms of CT education, there are problems with teacher expertise, assessments and questions over whether CT and CS should be a major part of curricula along with mathematics and science. CT has been used as justification by policymakers for CS being taught in primary education, but lack of teacher expertise and formal assessments could be harming student attitudes and learning outcomes. These problems are made worse by questions of whether CT is a universal skill required by all children. Armoni (2016) and Nardelli (2019) argue that CT should be seen as an explanation of the benefits of CS, not as a new discipline or separate subject. Just as mathematical thinking and linguistic thinking are not removed from mathematics or linguistics. Educators require the support of well-tested and age-appropriate programming tools. These tools should give children a basic understanding of CS, whilst being fun and engaging.

Despite the issues surrounding CT, some studies suggest that simple algorithmic programming concepts such as ‘using sequences of steps’ can transfer to other tasks. Kazakoff, Sullivan & Bers (2012, 2014; 2013) found that computer programming using a robotics kit improved story sequencing ability in children age 4 to 7. The next chapter (Chapter 4) describes an exploratory study that aimed to reproduce those results using a programming game with children age 5 and 6.

Chapter 4

Study 1 - Measuring the Effect of a Programming Game on Story Sequencing Ability in Young Children

Chapters 2 and 3 discussed the theoretical and practical limitations of teaching computer science (CS) and computational thinking (CT) in primary education. Children begin their CS education by being taught to understand, create and predict the behaviour of simple algorithms (Section 2.2.2). Algorithm prediction requires the learner to 'trace' code: mentally simulating the program step-by-step to predict the outcome. Lister (2011) suggests that novice programmers progress from manipulating code using trial and error to successfully tracing relying on specific values and finally being able to understand portions of code simply by reading (no tracing required). This type of algorithmic thinking forms part of the working definition for CT given in Chapter 3, defined as 'using sequences of steps and rules to solve a problem.'

The action of 'sequencing' objects or actions is an important skill for young children to develop in mathematics and literacy. For example, ordering numbers in the correct sequence and retelling a story in a logical sequence. Moreover, constructing narrative scripts or sequences of daily routines is common in early childhood curricula (Paris & Paris, 2003). Kazakoff & Bers (2012) argue that computer programming can be seen as a type of story sequencing. They conducted a series of experiments to see if story sequencing could be improved by teaching basic programming to young children (age 4 to 7), finding that participants improved on a story sequencing task after a programming intervention.

The effect of programming on story sequencing is a significant finding because there is limited evidence that CS and CT can improve skills on non-computational tasks (Denning et al., 2017). This chapter describes an exploratory study that was designed to reproduce these results using a programming game. It starts by explaining the studies and their rationale before describing several weaknesses and corresponding new methodological changes. It then moves onto the method, results and discussion of this study.

4.1 Programming and Story Sequencing

4.1.1 Rationale

Sequencing is the action of putting objects or actions into the correct order (Zelazo, Carter, Reznick, & Frye, 1997). It is an important part of early childhood curricula (age 3 to 8) in both mathematics and literacy; for example, putting words, letters and numbers in the appropriate order (Department for Education, 2013). Using sequences of pictures for storytelling is a common task for this age group as it requires sequential thinking and narrative understanding without relying on words (Paris & Paris, 2003).

Kazakoff & Bers (2012) argue that computer programming is a version of story sequencing: symbolic commands are arranged in an appropriate sequence to tell a computer what to do (Liao & Bright, 1991), with programmers thinking in sequential terms of next, before and until (Pea & Kurland, 1984). From this, the authors hypothesised that children who engage in programming activities would increase their story sequencing skills.

4.1.2 Studies

Kazakoff, Sullivan & Bers have reported three separate studies to measure the impact of programming on story sequencing, building on a laboratory-based pilot study (2011) (a full report on this study is the last one described in this section (2014)).

This section will discuss the methodology and results of each study in the order they were published. The following section will critique the studies and suggest methodological changes to address these weaknesses.

Kazakoff & Bers (2012)

The first study involved 54 children and was conducted in two schools (one private and one public), with an experimental and control group in each. Teacher experience using technology varied between the schools. Group 1 contained 22 children (64% male and 32% female) with an average age of 5.65 (*SD* = 0.39), who were divided evenly into an intervention group and a control group. Group 2 was comprised of two classes used as separate groups, 15 children (60% male and 40% female) in the intervention class with an average age of 5.54 (*SD* = 0.33) and 17 children (59% male and 41% female) in the control class with an average age of 6 (*SD* = 0.27).

Children in the intervention groups were exposed to the TangibleK program (now called KIBO) for 20 hours, taught by the class teacher. The control groups did art activities during this time. Story sequencing skills were assessed at pre-and post-test using Baron-Cohen, Leslie and Frith's (1986) assessment described in Section 4.2.3. The assessment used by Kazakoff & Bers contained five story sequences for the participant to arrange, with a maximum score of 10 points.

TangibleK involves a developmentally appropriate programming language called CHERP (Creative Hybrid for Robotics Programming). CHERP uses either cubes of wood covered in pictures depicting units of code (e.g. forwards, backwards, turn right, turn left) or on-screen instructions of the same blocks. The physical blocks are then converted to on-screen instructions that are then executed by a robot.

The authors found that children in the intervention groups improved their scores on the sequencing assessment compared to the control groups. Interestingly, the control group scores fell between pre-and post-test (Table 4.1). The authors give natural fluctuation between 'pre-operational' and 'concrete' thinking for children of this age as a possible reason for this (Section 2.2.2).

Table 4.1: Sequencing assessment scores in the first study by Kazakoff & Bers (2012)

Classroom	Type	*N*	Pre-Test Average	Post-Test Average	Change	% Change
1A	Private Intervention	11	7.55	8.82	1.27	17%
1B	Private Control	11	7.82	6.91	-0.91	-12%
2A	Public Intervention	15	7.4	7.6	0.20	3%
2B	Public Control	17	8.53	7.59	-0.94	-11%

Kazakoff, Sullivan & Bers (2013)

In the second study, Kazakoff, Sullivan & Bers (2013) repeated the experiment to see if they could achieve the same result with a 1-week intensive programming intervention (10 hours total). The participants were 27 children in either pre-kindergarten (age 4 and 5) or kindergarten (age 5 and 6). Once again, the experiment used the CHERP programming language combined with Lego WeDo Robotics Construction Set. The participants of the intervention group attended a public, early childhood Engineering magnet school. Whilst the control group were 13 children from a small, university-affiliated childcare centre. The assessments were given either side of the 1-week intervention.

The results are shown in Table 4.2. The authors found that kindergarten children in the intervention group improved significantly, $t(13) = 4.84$, $p < .001$. That children in the pre-kindergarten group also improved significantly, $t(12)$, $p < .05$. But that children in the control group did not improve, $t(13) = 0.291$, $p = .78$.

Table 4.2: Sequencing assessment scores in the second study by Kazakoff, Sullivan & Bers (2013)

Type	Age	*N*	Pre-Test	Post-Test
Intervention	5-6 (kindergarten)	14	6.43	8.79
Intervention	4-5 (pre-kindergarten)	13	3.77	4.45
Control	4-6	13	6.07	6.36

Kazakoff & Bers (2014)

The third study aimed to control for confounding variables that arise within the classroom, such as collaboration between children. Participants in the study had 3 one and a half-hour structured sessions in a laboratory working one-on-one with a researcher, as well as 1 initial one and a half-hour group session with four participants and three researchers. As in the first two studies, the participants used the CHERP programming system and a robotics toolkit. The participants were 34 children (64% male and 32% female) age 4 to 6. They had been recruited through flyers and emails sent to local elementary schools.

The mean pre-test score was 7.06 (*SD* = 2.45) and the mean post-test score was 8.44 (*SD* = 1.76); $t(33) = 2.71$, $p < .01$. The average time between pre-and post-test was 17.8 days (*SD* = 5.7). There were four perfect scores on the pre-test and one participant scored 0 on the pre-test and 10 on the post-test. The authors reanalysed the data without these participants and still found a statistically significant difference.

4.1.3 Critique and Methodological Changes

Despite the indications that computer programming improves story sequencing, there are several issues with the studies described in the previous section. This section describes three of these issues and how they will be addressed in the new study.

Inactive Control Groups

The first weakness of the studies is the use of inactive control groups. Participants in the experimental group may have changed their behaviour compared to the control because they were being observed during the intervention. This tendency to alter behaviour when being watched is known as the Hawthorne effect (Roethlisberger & Dickson, 1939). Yet, Shipstead, Redick & Engle (2012) use it to refer to any psychological phenomena that are introduced when intervention and control groups are treated differently. In these studies, participants in the intervention were introduced to technology and programming concepts that were both exciting and unusual compared to the experience of the control groups. This difference in treatment is particularly important because of the young age of participants.

One method of controlling for Hawthorne effects is to use an active control group, who are given a different activity during the experiment to keep them engaged. Having an active control "makes the experience of the participants in the baseline condition more comparable to those in the intervention condition, potentially equating the social contact experienced during the training period and reducing motivational differences between the groups." (Simons et al., 2016, p. 116). This can help clarify that the intervention is responsible for any cognitive improvements. In the study that did use an active control group, these participants did 'art activities', which is quite different from the exciting and unusual technology that the intervention groups were using.

To address this concern, the new study will have an active control group that uses a phonics application, also on tablets, that does not require the same step-by-step sequencing as the computer programming intervention task.

Sequencing Assessment

Secondly, the sequencing assessment used in the studies resulted in high average scores, often above 65% at pre-test and 75% at post-test. Kazakoff & Bers (2014) also state that there were ceiling and floor effects at both pre-and post-test in one of the studies. Dimitrov & Rumill (2003) suggest that ceiling effects indicate an easy test which falsely favours low-ability participants. The assessment was conducted one participant at a time, with a researcher delivering vocal instructions and explaining parts of each story sequence if required. This method of delivery could have resulted in differences between participants, for example, if one child had a further explanation that another child needed but did not ask for.

The sequencing assessment for the new study will be delivered using a computer program to reduce differences in delivery between participants. It will have three times as many questions and a time limit to increase the spread of results and reduce ceiling effects.

Control Groups from Different Institutions

The third weakness with two of the studies is either having no control group (Kazakoff & Bers, 2014) or a control group from a different institution (Kazakoff et al., 2013). In the second study, the participants were from a public magnet school in the Harlem area of New York City, whereas the control group "were part of a small, university-affiliated child care center outside of Boston, MA" (Kazakoff et al., 2013, p. 249). There is no guarantee that these groups are from a comparable sample of the population, which questions the validity of the control. However, it is worth noting that the first study did have both intervention and control groups in each institution (Kazakoff & Bers, 2012). Yet, the sample size for this study was small and one of the intervention subgroups did not improve on the task (Table 4.1).

The new study will use a control group comprised of children from the same institution and randomly assign them to the intervention condition or the control.

4.2 Method

This was an exploratory study to see if the repeated findings of Kazakoff, Sullivan & Bers (2012, 2014; 2013) could be replicated using an age-appropriate programming game, Lightbot Jr, instead of the physical CHERP programming language and robotics toolkit. The study used the methodological changes described in the previous section (Section 4.1.3): an active control group, an automated and longer sequencing assessment and a control group from the same institution.

4.2.1 Participants

Participants in this study were 50 children age 5 and 6 (*M* = 6.2, *SD* = 0.27) from a large primary school in northern England. The sample comprised of 40% male and 60% female participants. The school is above the national average of pupils meeting the expected standard in reading, writing and maths with 63%. The original sample contained 60 children of an even gender split, however, some have been excluded from data analysis due to being absent during parts of the intervention or the post-test.

4.2.2 Design

The study followed a pre-test post-test experimental design to measure for improvements on the story sequencing assessment after playing a programming game (Figure 4.1). The participants were split into two groups: intervention (Lightbot Jr) and active control (Twinkl Phonics). Participants were assessed using a story sequencing assessment adapted from Baron-Cohen, Leslie and Frith (1986).

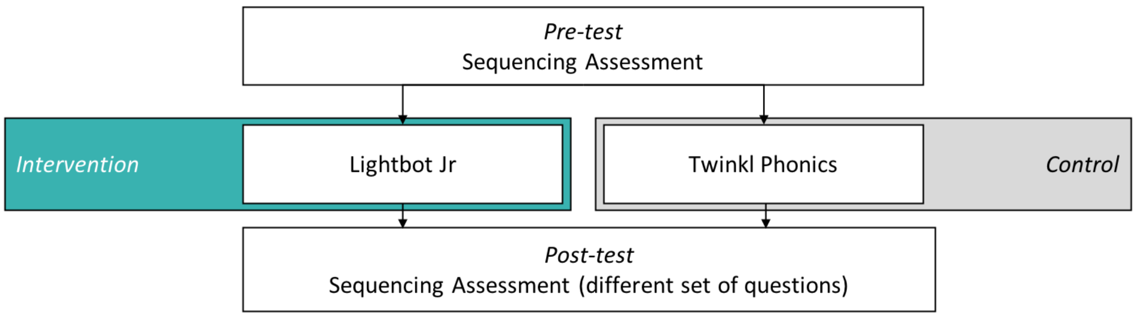


Figure 4.1: Diagram of the Study 1 design

4.2.3 Materials

Sequencing Assessment

The sequencing assessment was adapted from picture sequencing cards created by Baron-Cohen, Leslie and Frith (1986) for a study comparing high-ability autistic children to low-ability Down's syndrome children. This is a similar assessment to the one used by Kazakoff, Sullivan & Bers, with 20 more story sequences added, resulting in six stories from each of five the categories listed below. Each story contains four picture cards that must be ordered correctly for the story to make sense. An example story is shown in Figure 4.2 and all the sequences are shown in Appendix E.

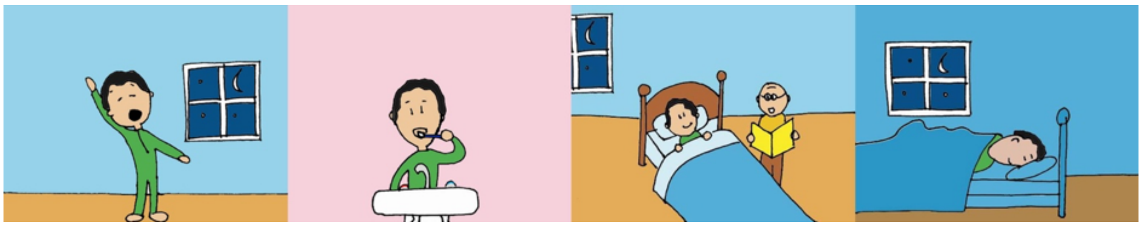


Figure 4.2: The 'Going to bed' story sequence

These stories were broken down into the five categories originally used by Baron-Cohen, Leslie and Frith. These were used in the study as an indicator of difficulty:

1. Mechanical 1 (objects interacting causally with each other)
2. Mechanical 2 (people and objects acting causally on each other)
3. Behavioural 1 (a single person acting out everyday routines)
4. Behavioural 2 (people acting in social routines)
5. Intentional (people acting in everyday activities requiring the attribution of mental states)

A software application was created to present the story sequences using the standardised procedure created by Baron-Cohen, Leslie and Frith (1986) (Figure 4.3). The first card in the sequence is placed in the correct location, the other cards are placed above it in a random order, correcting for the child spontaneously placing cards in the correct order. The participant then selects a card and places it in its appropriate position in the sequence. Cards can be moved back to the top of the screen by selecting them again. This interaction method allows children who struggle with a computer mouse to use the application effectively. When the participant is happy with their answer, they select the 'Finished!' button and a new sequence is shown.

Participants received 2 points for a correct sequence, 1 point for the correct beginning and end card, and 0 points for an incorrect sequence. They had 4 minutes to order as many stories as they could out of a set of 15. The maximum score was

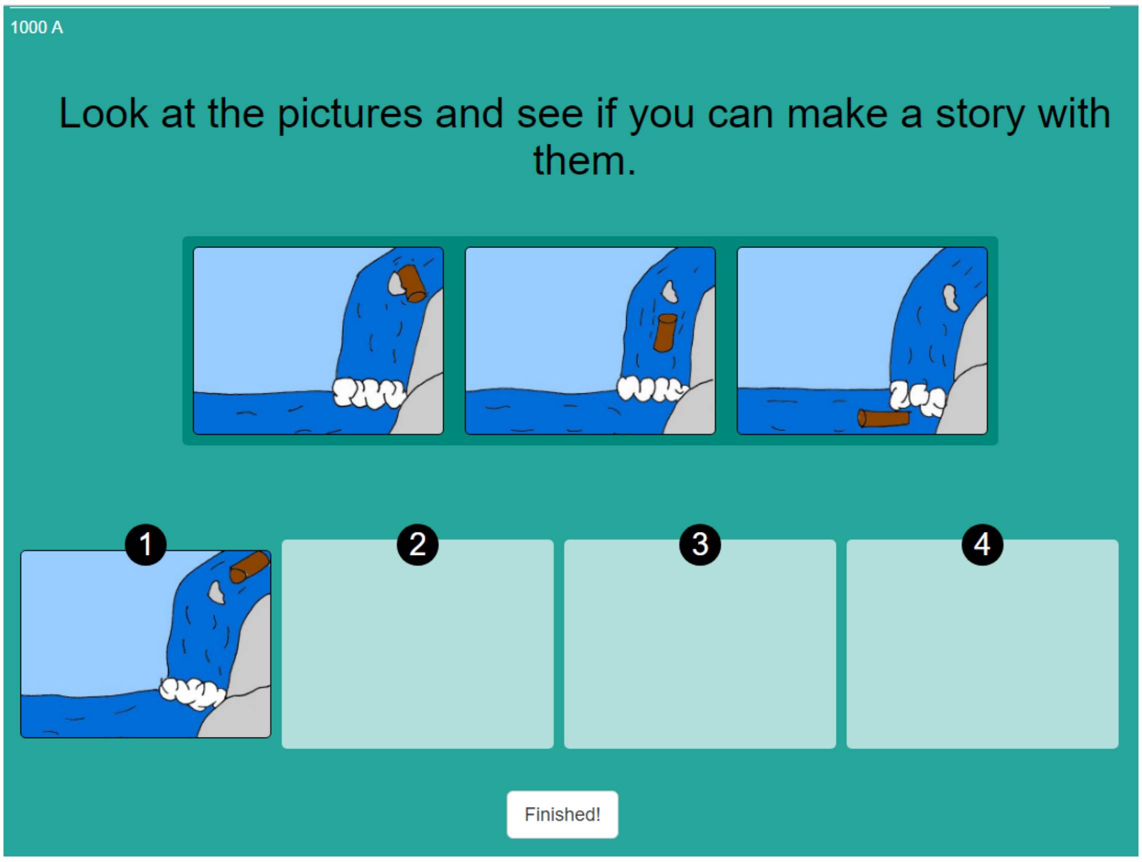

1000 A
Look at the pictures and see if you can make a story with them.
1
2
3
4
Finished!

Figure 4.3: Screenshot of the sequencing assessment application

30 points. Two sets of 15 story sequences were produced, containing the same number of stories from each category so that they were of similar difficulty. These were alternated for each participant, meaning that one participant would do the first set for the pre-test and the second for the post-test, then the next participant would do the second for the pre-test and the first for the post-test and so on. The stories were presented in a random order, which along with alternating the sets of questions, was done to reduce the likelihood of participants copying each other and unintended differences between the difficulty of each set.

The application was tested in two other schools with children of the same age before the study as part of an iterative development process. These observations resulted in several important changes in the final application:

1. Allowing the user to operate the application using clicks, instead of clicking and dragging with the mouse, which some children found difficult.

2. Setting the time limit to 4 minutes so that not all children would finish the 15 stories, meaning there would be a bigger spread of results. There was no time limit during testing to see how long it took each child to complete all the stories.

3. Bigger images and text as well as numbered spots for card position in the sequence.

Lightbot Jr

Lightbot Jr (Section 2.4.5) was chosen for the intervention as it is age-appropriate, can be played on a tablet and requires the player to program by placing instructions in sequence (Figure 4.4). The goal of each level is to program the robot to turn all the blue spaces in a level into illuminated yellow spaces. This is done by arranging symbolic instruction blocks (forward, turn left, turn right, jump and lightbulb) that tell a robot what to do, similar to CHERP.

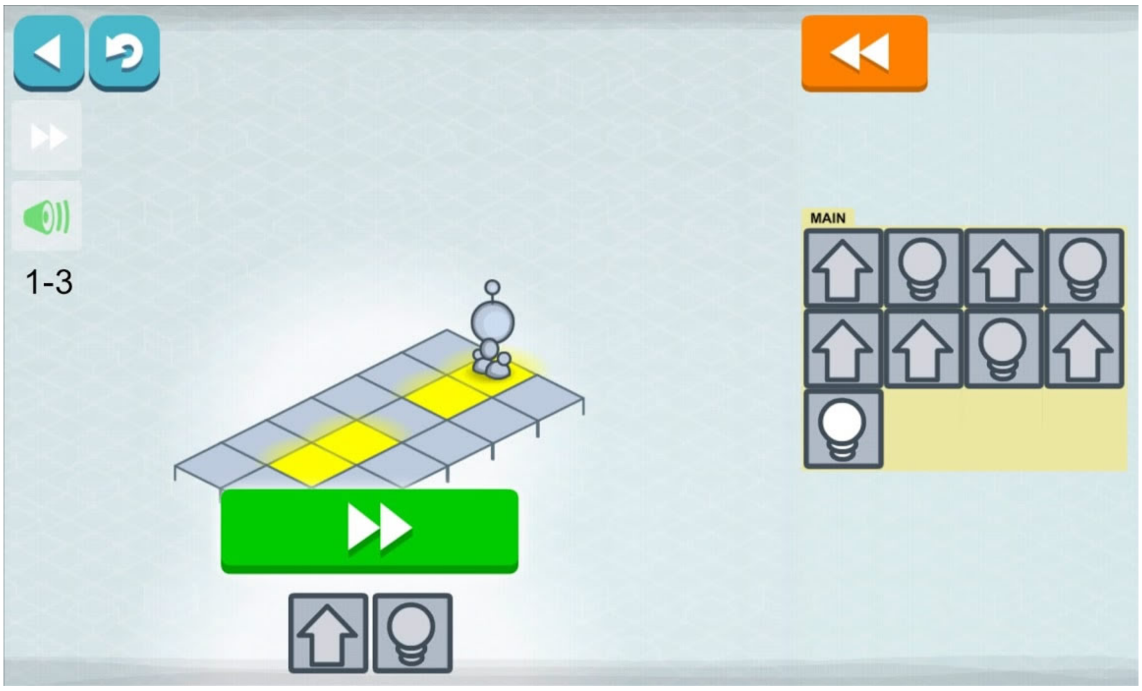


Figure 4.4: Screenshot of Lightbot Jr

Twinkl Phonics Suite

The active control group used the Twinkl Phonics Suite (Figure 4.5), an application that contains a range of phonics-based activities, including sounds and names of letters, letter formation, blending sounds and high-frequency words. It was chosen as an alternative to Lightbot Jr because it does not contain the same step-by-step sequencing where symbolic instructions are placed in order. The school specified that they would like it used as it had a wide range of activities relevant to participants' classroom learning outcomes to keep them occupied for the duration of the study.

4.2.4 Procedure

The sequencing assessments took place in the school IT suite. Participants were given a demonstration beforehand to explain what they would be doing and how to use the application. After the pre-test, participants were randomly assigned to either the intervention or control condition using a matched-pairs design.

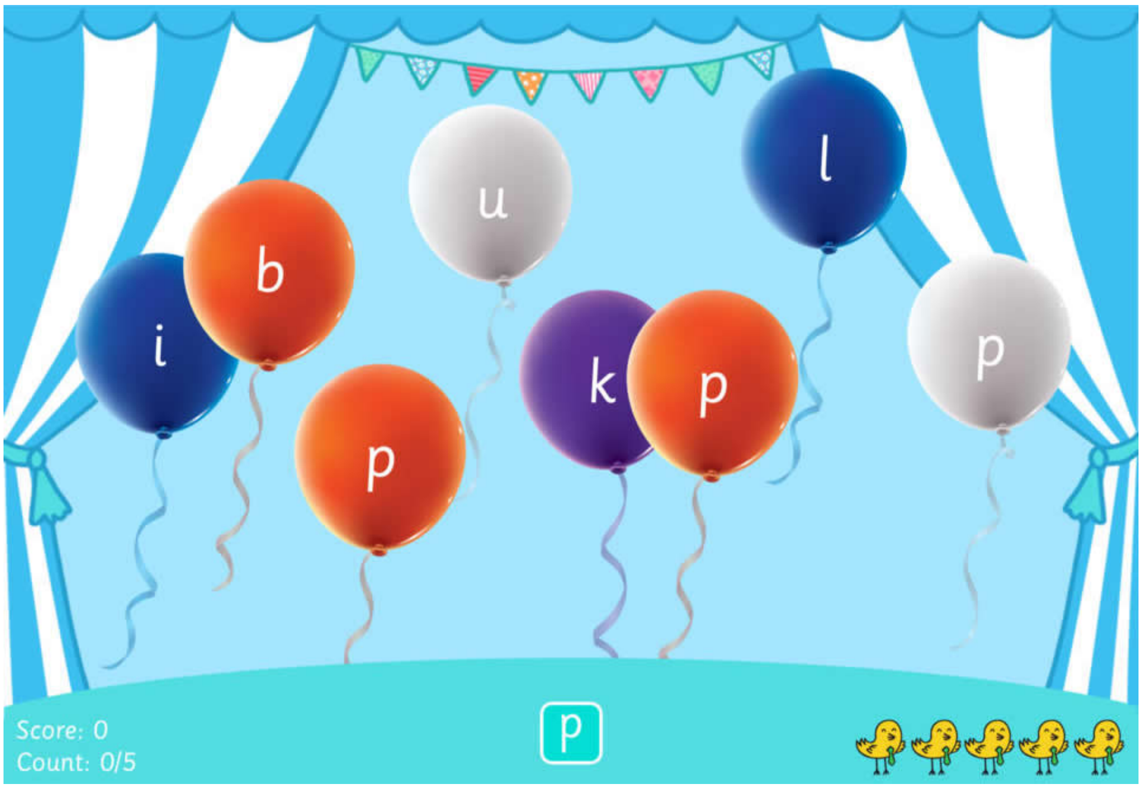


Figure 4.5: Screenshot of a task from the Twinkl Phonics Suite where the player selects balloons containing the same letter as the one at the bottom of the screen

Participants then used either Lightbot Jr or the Twinkl Phonics Suite in groups of 10, for 20 minutes a day in class time over the school week (five days, 100 minutes total). This took place in a small intervention room situated between two classrooms. Each child was given a tablet for this period that was locked to the application they would be using. Participants in the intervention group used the same tablet each day so that they could continue where they had finished the previous day. Lightbot Jr progress and researcher observations were recorded at the end of each session to chart participant progress.

The post-test was completed 10 days after the pre-test, at a similar time of day for each group. The study ran between the 5th May 2017 and the 15th May 2017.

4.2.5 Ethics and Access to Participants

University approval was given for a series of studies investigating the effect of visual programming on CT skills (Appendix C). For this study, approval was acquired from the headteacher of the school. Opt-out consent forms were then sent to the parents or guardians of participants (Appendix F), in line with the school's wishes.

All data, including test scores, game progress and researcher observations were anonymised using participant ID numbers. Appendix D shows the data management plan.

4.2.6 Hypotheses

It was hypothesised that the study would replicate the repeated findings of the previous studies by Kazakoff, Sullivan & Bers (2012, 2014; 2013); children in the programming condition would improve from pre-to post-test on the story sequencing assessment, compared to the control.

In addition, it was expected that the sequencing assessment would produce a good range of scores without ceiling effects, that the sequencing assessment would be a predictor of Lightbot Jr progress (to give some indication that sequencing is related to programming) and that Lightbot Jr progress would be a predictor of learning gains from pre-to post-test.

4.3 Results

4.3.1 Effectiveness of the Sequencing Assessment

Both the pre-and post-test produced a good range of scores (Table 4.3). Figure 4.6 shows the distribution of the pre-and post-test scores. Histograms have been used here to identify ceiling effects, as one of the aims of the new sequencing assessment was to negate these. There was a potential ceiling effect at post-test, with five participants scoring within 5 points of the upper limit (30 points). Yet, the scores were normally distributed, with skewness of -0.28 (*SE* = 0.34) and kurtosis of -0.051 (*SE* = 0.66). Using a paired samples t-test, there was a significant improvement for the participants overall between the pre-and post-test ($t(49) = -4.33$, $p < .001$, $d = 0.57$).

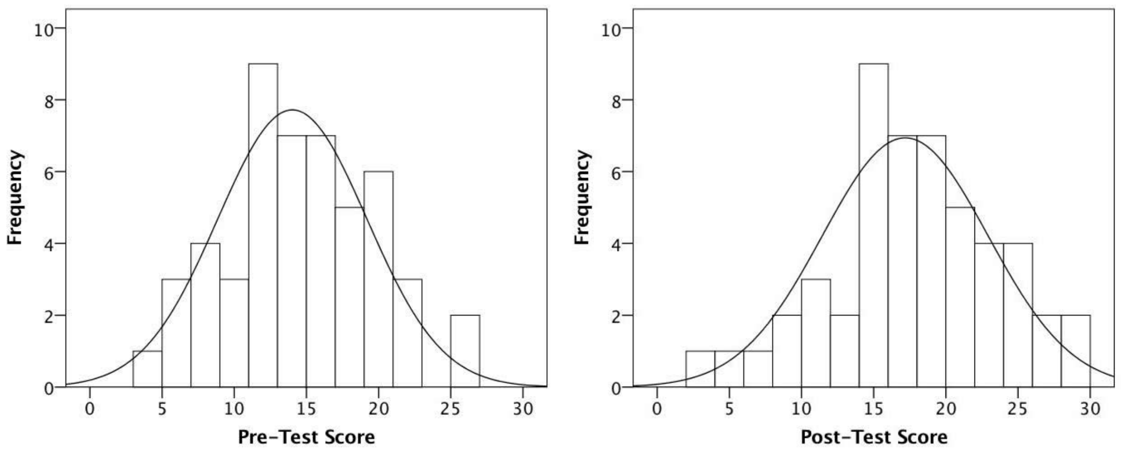


Figure 4.6: Histogram of the sequencing assessment spread of results at pre-and post-test

Table 4.3: Descriptive statistics for the sequencing assessment at pre-and post-test

	Pre-Test	Post-Test
N	50	50
M	14.02	17.18
SD	5.17	5.75
Minimum	4	3
Maximum	25	28

4.3.2 Differences in Story Sequencing

A one-way ANCOVA was conducted to test for differences on the pre-and post-test between groups. This method used the post-test scores as the dependent variable and the pre-test scores as a covariate, assessing for differences in the post-test means after accounting for pre-test values (Dugard & Todman, 1995). The difference between groups was not significant; $F(1, 47) = 0.03$, $p = .86$, $\eta^2 = .001$. Figure 4.7 shows the average learning gains for each group. Further statistics are provided in Table 4.4.

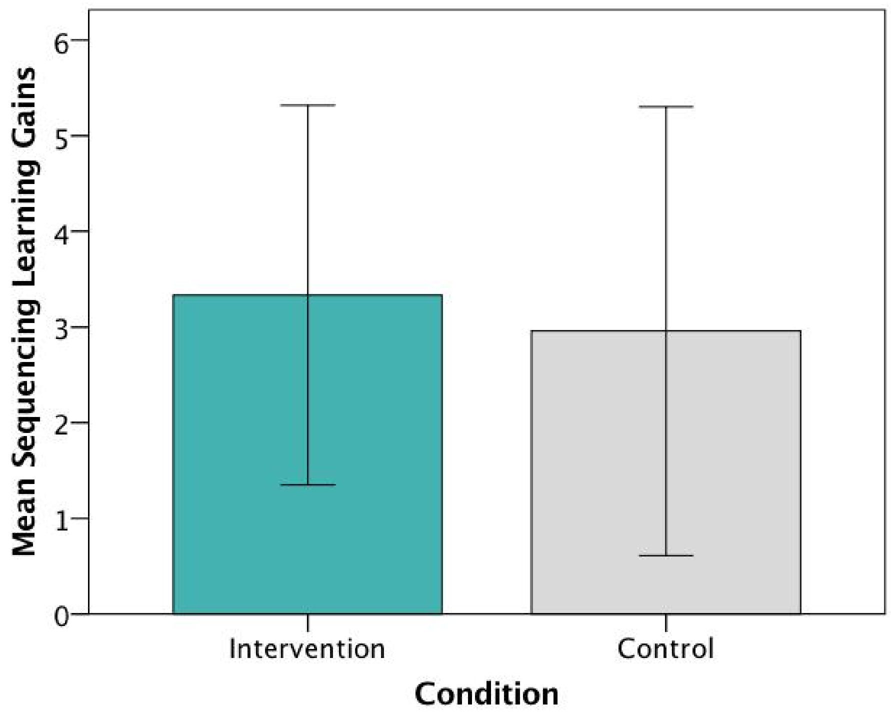


Figure 4.7: Comparison of the average learning gains on the sequencing assessment from pre-to post-test for each group (error bars show 95% confidence interval)

4.3.3 Sequencing as a Predictor of Lightbot Jr Progress

There was a correlation between a participant's pre-test score and their number of Lightbot Jr levels completed, $r(50) = .46$, $p = .015$ (Figure 4.8), indicating that the story sequencing pre-test was a predictor of Lightbot Jr performance.

Table 4.4: Descriptive statistics for the sequencing assessment at pre-and post-test for each group

Condition		Pre-Test	Post-Test	Learning Gains
	M	13.85	17.19	3.33
Intervention	*N*	27	27	27
	SD	5.70	5.84	5.02
	M	14.22	17.17	2.96
Control	*N*	23	23	23
	SD	4.59	5.77	5.42

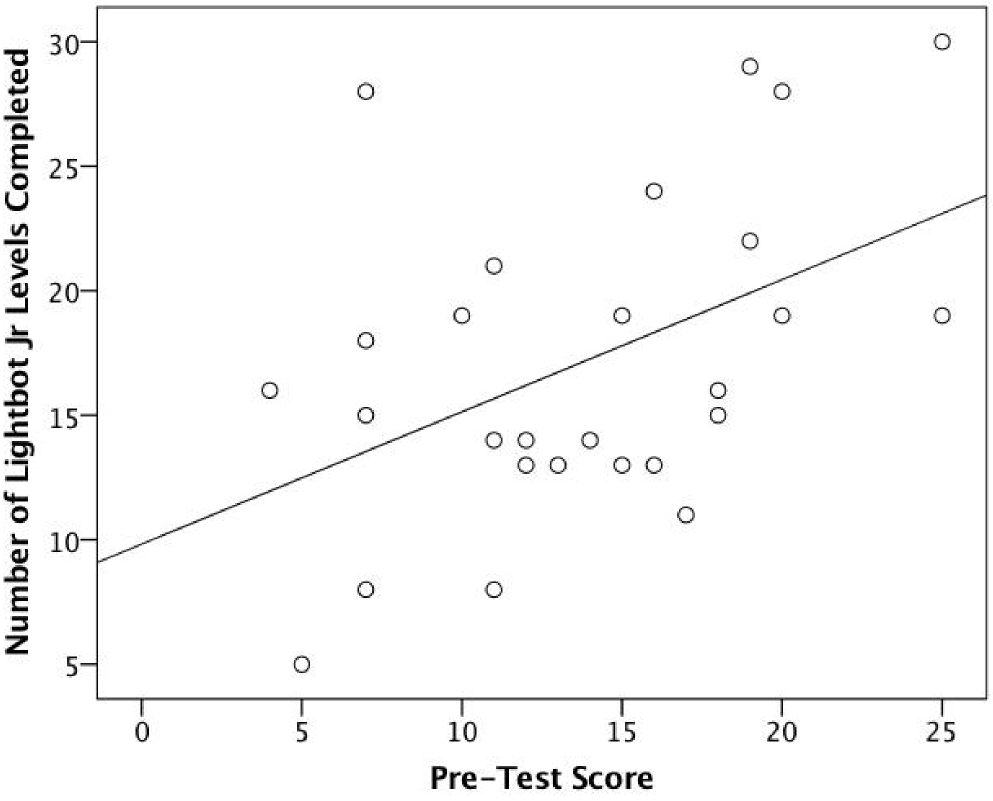


Figure 4.8: Relationship between pre-test sequencing assessment score and Lightbot Jr progress (with regression line)

4.3.4 Lightbot Jr Progress as a Predictor of Learning Gains

There was a negative correlation between Lightbot Jr levels completed and learning gains between pre-and post-test, $r(49) = -.42$, $p = .02$ (Figure 4.9). Children who did not progress as far in Lightbot Jr improved more on the story sequencing assessment.

4.4 Discussion

In summary, the sequencing assessment produced a good range of scores (21 in the pre-test and 25 in the post-test). However, there was a ceiling effect in the post-test, which suggests that the test may have been too easy. The intervention and the active

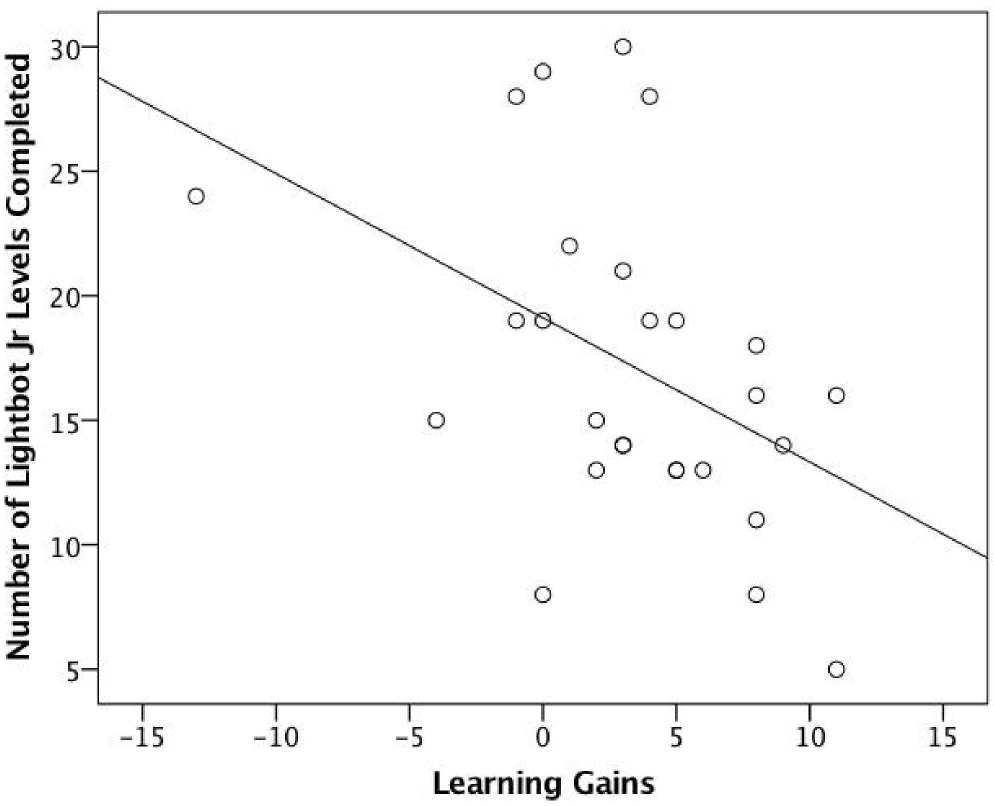


Figure 4.9: Relationship between Lightbot Jr progress and learning gains on the sequencing assessment (with regression line)

control both improved on the story sequencing task, with no difference between the groups. The pre-test scores were a predictor participant progress in Lightbot Jr, but this progress was, in turn, a predictor of lower learning gains between the pre-and post-test.

The finding that both the intervention and control groups improved on the sequencing assessment is at odds with the previous studies by Kazakoff, Sullivan & Bers. Possible reasons for this include the limitations in those studies described in Section 4.1.3 (e.g. lack of an active control group), the link between sequencing and programming is not as strong as expected and limitations with the methodology of this study (problems with Lightbot Jr, sample size and intervention length). This section discusses the link between sequencing and programming and the limitations of this study in more detail.

4.4.1 The Link Between Story Sequencing and Programming

Story sequencing may not be as intrinsically linked to programming as first expected. Likely, the improvements in this study for both groups from pre-to post-test on the story sequencing assessment can be attributed to the practice effect: participants improved because they were used to the format and style of the assessment. The correlation between pre-test scores and Lightbot Jr progress does suggest that story sequencing is in some way related to programming, yet this could be far transfer that requires further mediation.

Denning, Tedre & Yongpradit (2017) argue that everyday step-by-step procedures (similar to the story sequences used in this study) are not the same as machine-executable algorithms used in programming (Chapter 3). Therefore, it may be that Kazakoff, Sullivan & Bers' argument that computer programming is a version of story sequencing is misguided. Despite programmers thinking sequentially and using 'sequences of steps and rules to solve a problem', it could be that the jump from this to sequencing stories of 'human steps' (the isolated actions of a person) is too large. This highlights questions of transfer between computational and non-computational domains. In experiments done with the Logo programming language, researchers found that problem-solving skills developed through programming did not transfer to non-computational contexts (e.g. Clements & Gullo, 1984; Pea & Kurland, 1984). The problem of CT skills like algorithms and sequencing transferring to non-computational tasks (like story sequences) was discussed in criticisms of CT section in the previous chapter (Section 3.4). An additional measure of algorithmic sequencing or programming ability could be used in future studies to test whether story sequencing and programming are related as the correlation between pre-test scores and Lightbot Jr progress suggests.

The negative correlation between Lightbot Jr progress and learning gains could be explained by higher-scoring participants already possessing sequencing skills before the intervention. This meant that there was simply less room for improvement in the post-test, because of the ceiling effect, compared with lower-scorers.

4.4.2 Lightbot Jr Understanding

Some participants struggled with aspects of Lightbot Jr, despite the game being designed for children age 4 to 7. This lack of understanding may have resulted in participants not developing programming skills as expected during the intervention.

Instruction Overlay

It was observed that participants ignored the visual instructions at the start of levels, choosing instead to ask researchers for an explanation. Most participants were not confident readers and may have also avoided instructions because they overlay the game itself, meaning the player can see the game waiting to be played behind them (Figure 4.10).

Debugging Difficulties

Participants had difficulty identifying and removing incorrect blocks from their solutions. They often chose to remove all instructions from the program and start again

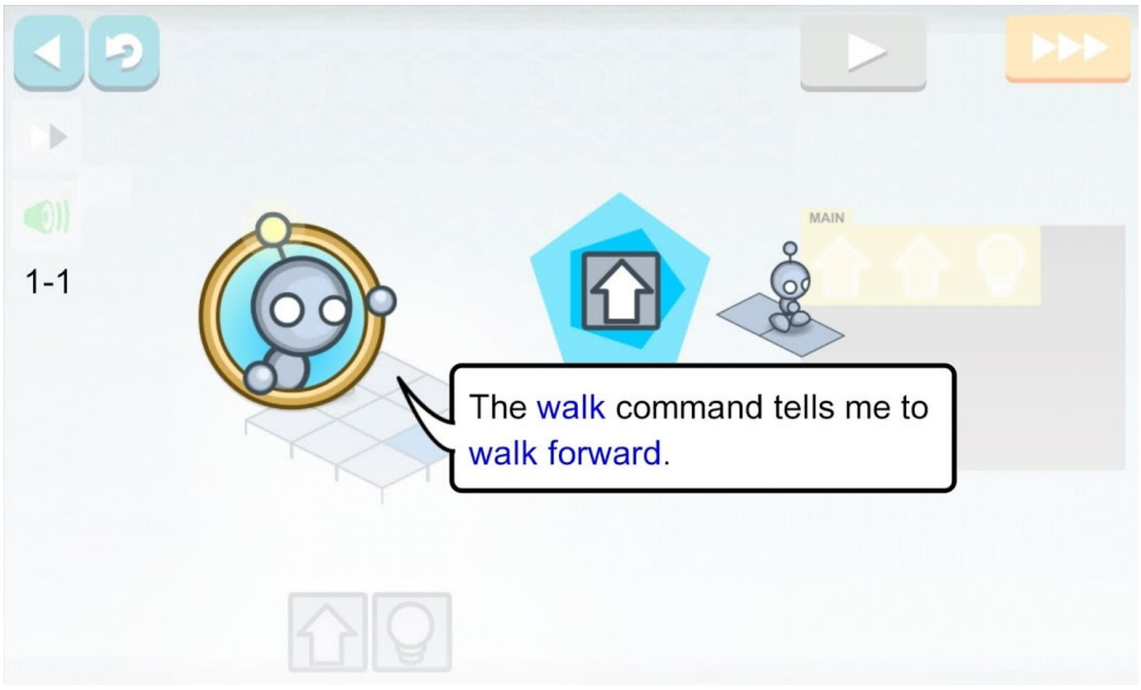


Figure 4.10: Screenshot of Lightbot Jr instructions with the game visible behind

rather than attempting to debug their solution. Participants would also attempt to complete the level in one go, without breaking down the problem into parts. This lack of debugging and decomposition could be due to underdeveloped working memory: participants could not hold the executing instruction and movement of the robot in their heads simultaneously. The difficulty with debugging, decomposition and abstraction may be a limitation of teaching programming to this age group, echoing concerns of Armoni (2012). This supports Lister's (2011) theory that novice programmers initially struggle with the abstract relationship between different parts of the code and can only focus on one instruction at a time.

This study required participants to understand the sequencing used in programming and then to transfer these skills to another task. Whilst there is evidence that children as young as age 4 can understand basic programming concepts (Bers, 2010; Fessakis et al., 2013), it may be that Lightbot Jr is too difficult for children without prior knowledge. As previously mentioned, this could be addressed by using a measure of programming ability, in addition to story sequencing, to test if participants have developed programming skills as expected.

4.4.3 Sample Size and Intervention Length

The lack of a between-groups difference in this study could be because of the sample size. Yet, given that this was an exploratory study designed to test the assumptions of Kazakoff, Sullivan & Bers, it did use a similar sample size to their studies and found no difference. Moreover, due to the effect size of the between-group difference (η^2 = .001), the new study would need 7843 participants to show a significant improvement in story sequencing compared to the control (using a type 1 error rate of 0.05 and type 2 error rate of 0.8). The small effect size suggests that such a study would not be worth doing.

Another limitation of this study was the length of the intervention. Classroom logistics meant that participants only had 100 minutes of programming during the intervention, compared to the 6, 10 and 20 hours in Kazakoff, Sullivan & Bers' studies. However, a much larger study by the National Foundation for Education Research found that a year-long programming intervention using Scratch did not affect the CT ability (measured using Bebras (Section 3.3.2)) of 317 children age 9 to 11 (Straw et al., 2017). Furthermore, Boylan et al. (2018) found that mathematics outcomes of 5,818 children age 9 to 11 did not improve after a year (or two years in some cases) of Scratch programming, but that CT scores using a different subset of Bebras tasks did. These results, along with the results of this study, suggest that measuring the impact of programming on CT and other cognitive skills (such as story sequencing) is difficult and that additional measures or mediation is required.

4.5 Conclusions

In conclusion, there was no difference observed between the effects of a programming game and phonics activities on story sequencing ability in children age 5 and 6. The overall improvement of both groups could suggest that the findings of Kazakoff, Sullivan & Bers have been influenced by methodological design weaknesses, such as using an inactive control group, a short and manual assessment or a control group from a different institution. This raises concerns as to whether their improvements in story sequencing can be attributed solely to the programming intervention. Yet, due to differences in the intervention and limitations of this study, their results cannot be dismissed outright.

The cross-disciplinary application of computer programming for young children is an interesting area. CT literature argues that skills developed through programming are useful in other subject areas. However, there is no guarantee that children in early childhood can develop an understanding of these concepts and limited valid and reliable CT assessments, particularly designed for younger children. This study supports the issues with CT in primary education raised in Section 3.4. Furthermore, programming observations from this study echo the concerns that more abstract CS and CT concepts are too difficult for children under age 7 to understand and apply (Armoni, 2012).

4.5.1 Moving Away from Computational Thinking

The results of this study suggest that it can be difficult to measure for transfer and improvements of the CT concepts learnt through programming. In this case, the transfer of algorithmic sequencing to ordering story sequences. In addition, it has highlighted issues with teaching programming to young children (under age 7), in terms of their ability to formulate and debug programs.

This study has highlighted the ‘computational’ aspect of the working definition given in Section 3.2.4. It supports views that it is difficult to separate CT and CS (Armoni, 2016; Nardelli, 2019), particularly without CT measures designed for children this young. It makes sense, then, to focus on measurable aspects of CS and CT that older children can learn to use in existing educational programming tools, which are computational in nature. These can then be supported by existing CT measures, such as Dr. Scratch and the Computational Thinking test. This, in turn, means moving away from younger children, who lack the working memory to understand more abstract CS concepts, to children at upper-primary level (age 9 to 11).

Novices can experience problems and misconceptions when learning to program. Research suggests that they can form ‘bad programming habits’ in block-based tools that can make programs difficult to understand, debug and maintain. The following chapter (Chapter 5) discusses the concept of abstraction in more detail and how it can be used to teach novice programmers to correct bad programming habits by ‘smelling’ their code and learning to ‘refactor’ their programs using abstraction.

Chapter 5

Abstraction and Code Smells

Chapter 4 showed that it can be difficult to measure improvements and transfer in even the simplest aspects of computational thinking (CT). In this case, from algorithmic sequencing of instructions in a programming game to ordering story sequences. This fits with the perspectives of Armoni (2016) and Nardelli (2019) in suggesting that it is difficult to separate CT and computer science (CS), particularly without valid and reliable CT measures. Section 3.3 highlighted the fact that many existing CT measures have little or no evidence of validity or reliability. This is because CT is a relatively new field and producing reliable measures is a time-consuming process. Academics are playing catch-up to policymakers who have introduced CS and CT into national curricula without testable learning outcomes, adequate training resources and teacher support. As such, children are being taught CT and CS in primary education by teachers that lack expertise and confidence in CS, often using programming tools with little direct guidance (Webb et al., 2017). Until it is possible to reliably assess CT, it makes sense to focus on measurable aspects of CS (and therefore CT) that address common problems and misconceptions that can arise when primary school children learn to program. This also means concentrating on older primary school children (age 9 to 11), who use more complex programming tools (e.g. Scratch).

Chapters 2 and 3 identify abstraction as playing a key role in both CS and CT. Yet, these skills are not often taught in the primary school classroom, because they require expertise and understanding that teachers often lack (Rich et al., 2019). This is important because abstraction is a crucial part of writing ‘good’ code and can be used in popular block-based programming tools such as Scratch.

Scratch is the most widely-used programming tool in primary education, yet it’s constructionist, self-directed design means it can lead children to form bad programming habits (Meerbaum-Salant, Armoni, & Ben-Ari, 2011). These habits can make programs difficult to understand, debug and maintain. They can be addressed if the programmer is taught to ‘smell’ that something is wrong with their code and can ‘refactor’ the code to remove the smell. Common smells in Scratch include duplicated code, long scripts and dead code. The refactoring process requires good programmatic and procedural abstraction skills, which primary school children should be able to learn through structured teaching (Gibson, 2012).

This chapter discusses abstraction, which plays a key role in the thesis from this point in. It begins with its application in human cognition before explaining what it means in both CT and CS. The chapter then discusses the bad programming habits that can arise because of Scratch's constructionist design, before examining the different 'code smells' that indicate these habits and how abstraction can be used to remove them.

5.1 Abstraction

5.1.1 In Human Cognition

Abstraction is the conceptual process where general rules and concepts are derived from specific examples, literal (or 'concrete') signifiers or first principles. An 'abstraction' is the outcome of this process. Conceptual abstractions are formed by selecting only the aspects of an observable phenomenon that are relevant for a specific purpose. For example, abstracting a square to the more general idea of a 2D shape by selecting the relevant 'shape' information (area, perimeter, colour, etc.) and excluding the other characteristics that are only relevant to the square (e.g. four sides). Abstract reasoning is the ability to generalise about relationships and attributes as opposed to concrete objects, using categories, schemas and cognitive structures to organise and generalise information. Thinking in abstractions is one of the key characteristics of human behaviour (Piaget, 1970).

5.1.2 In Computational Thinking

Abstraction is the main tenet of CT (Wing, 2006) (Chapter 3). It forms part of the working definition of CT for this thesis and is defined, in combination with generalisation, as 'removing the detail from a problem and formulating solutions in generic terms'. In the CT definitions, frameworks and models analysed in Section 3.2, abstraction is discussed in the general context of recognising and generalising patterns (Shute et al., 2017), using procedures to encapsulate a set of repeated commands (Barr & Stephenson, 2011) and distinguishing CT from other types of thinking (Grover & Pea, 2013). Abstraction in CT is closely linked with generalisation, pattern recognition and decomposition.

There are several non-computational approaches used to teach abstraction to children. In one example from Barefoot (2019), who provide CT resources for primary and secondary education, children try to describe an animal by its features without using its name. This requires the child to only include what is important, therefore creating an abstraction. There is some evidence to suggest that tasks like this can improve novice confidence in programming (Hermans & Aivaloglou, 2017) and improve scores on the Computational Thinking test (Brackmann et al., 2017). However, there is little evidence that these skills can transfer to computational tasks, as discussed in Section 3.4.

5.1.3 In Computer Science

Abstraction is fundamental in CS and is “one of the most vital activities of a competent programmer” (Dijkstra, 1972, p. 864). The main objective of abstraction in CS is the process of ‘information hiding’ (Colburn & Shute, 2007): hiding, but not neglecting, details that are “essential in a lower-level processing context but inessential in a software design and programming context” (p. 176). The reason that information can be essential in one context and inessential in another is that tools for abstraction and information hiding have evolved over the history of software development. Programming languages, operating systems, network protocols and design patterns all allow programmers to operate at higher levels of abstraction.

Abstraction is also an important part of software engineering. The DRY (Don’t Repeat Yourself) principle states that duplication in logic should be eliminated via abstraction (Hunt & Thomas, 1999). Abstraction is used to model the problem domain: defining an object in terms of its properties, functionality and interface (how it communicates with other objects.) An ‘abstract’ class in object-oriented programming is one that cannot be instantiated but instead provides a base for subclasses to ‘extend’. In the earlier example in Section 5.1.1, ‘shape’ would be the abstract class that contains the relevant shape details (area, perimeter, colour) and ‘square’ would be a subclass that extends it with other details specific to the square (having four sides).

Duncan, Bell & Tanimoto’s (2014) heuristics used to categorise available programming tools in Section 2.4 use abstraction as a measure of complexity, with tools aimed at older children containing more abstract concepts like functions, variables and conditional execution.

Levels of Abstraction

Good computer scientists can move quickly and efficiently between levels of abstraction. Knuth described natural computer scientists as “individuals who can rapidly change levels of abstraction, simultaneously seeing things both ‘in the large’ and ‘in the small’ ” (Armoni, 2013, p. 266). Perrenet, Groote & Kaasenbrood (2005) defined a hierarchy of levels of abstraction in CS (referred to as the PGK hierarchy in the rest of the chapter):

1. Execution level – an interpretation of an algorithm as a specific run on a specific machine.

2. Program level – an algorithm as a process; a program written in a specific programming language.
3. Object level – an algorithm not associated with a specific language.
4. Problem level – considering the problem as an object, referring to its attributes and being able to deal with the solution to the problem as a black box.

At each of these levels, more information is hidden or excluded, with the programmer working at a higher level of abstraction. Perrenet, Groote & Kaasenbrood (2005) found that novice programmers at university level were working mostly at Program (2) and Object (3) level, with only a few working at Execution (1) or Problem (4) level. They also found that students were able to work at higher levels of abstraction the longer they had been on the course. This fits in with Lister's (2011) view that programmers develop abstract reasoning skill as they gain expertise. Statter & Armoni (2016) found that children age 13 and 14 could work at Object level (3) after a year of CS lessons focusing on abstract thinking, giving generalised verbal descriptions of algorithms in Scratch programs.

In Primary Education

Chapter 2 briefly discussed abstraction in CS for primary school children. Lister (2011) suggests that the development of abstract thinking in CS corresponds to being able to 'trace code' (mentally simulating the program step-by-step to predict the outcome.) New programmers require considerable effort to trace code and rarely manage to do so accurately. As they gain expertise, they can trace reliably using specific values but are unable to reason abstractly about the code. Eventually, they progress to tracing abstractly without using specific values and are then able to understand sections of code without tracing. Armoni (2012) used the limitations of novice programmers to question whether children can learn CS, particularly before age 7, going on to argue that in theory young children can learn abstract ideas through specific values, concrete objects and physical manipulation of information. This is the approach used by Gibson (2012) to successfully teach theoretical abstract CS concepts such as graph connectivity and graph isomorphism to children as young as age 5. Gibson suggests that children have the potential to learn abstract concepts in primary school, even from the point they can read and write, as long as the content is structured correctly.

However, Benton et al. (2018) suggest that primary school children can struggle with comparing different algorithms and tackling problems 'from above' (Object (3) and Problem (4) levels of the PGK hierarchy). In particular, problems that included procedures to abstract away information from the main program. Yet, many learners were able to give answers that suggested they could reflect on different programmatic strategies. Swidan, Hermans & Smit (2018) found that primary school children held programming misconceptions about variables, loops and conditionals that caused them to struggle to trace code in Scratch. This reiterates that structured teaching is required for primary school children to develop abstract reasoning skills in CS.

The next section explains how Scratch and other block-based programming tools can be problematic when teaching good CS practices, particularly those involving abstraction, to novices.

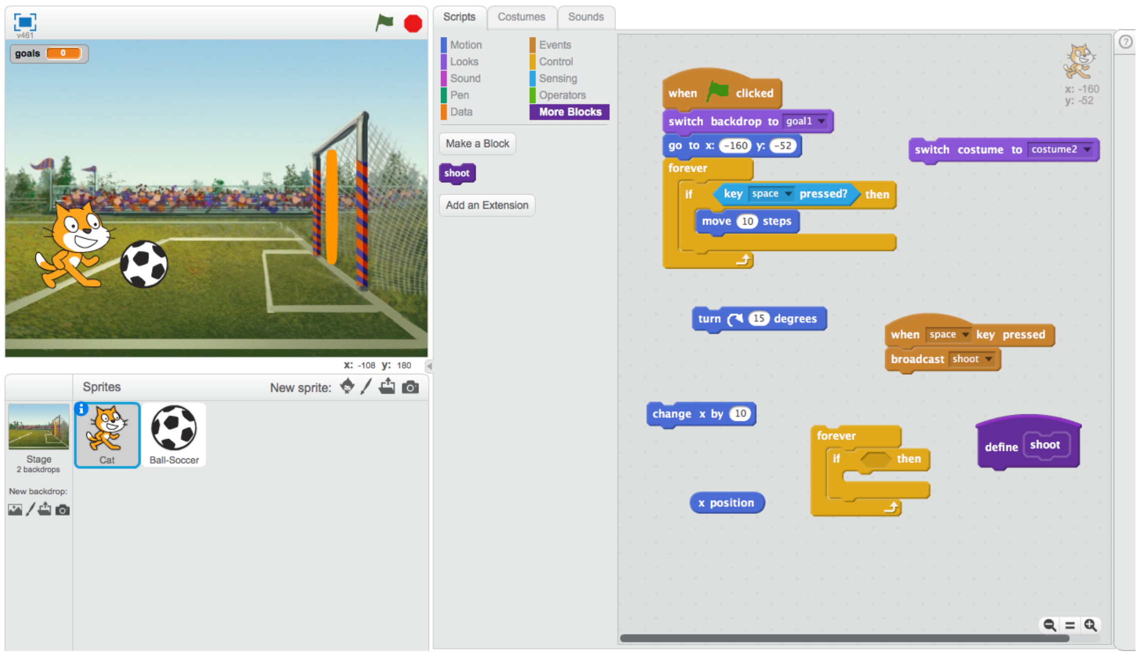
5.2 Code Smells

5.2.1 The Problem with Scratch

Abstraction skills correspond to a better ‘top-down’ approach to programming because the programmer can think about the problem at a higher level and better plan the solution before they begin to write code. However, Scratch encourages a constructionist ‘bottom-up’ (or ‘bricolage’) programming approach, where solutions are unplanned and created largely through exploration (Turkle & Papert, 1992). Scratch aims to “support self-directed learning through tinkering” (Maloney et al., 2010, p. 2). It inspires a bottom-up approach by making the block palette visible at all times, having little in-built guidance and feedback, giving no error messages and allowing ‘dead’ blocks to be in the program space that are not executed. These features are useful for the creative aspects of Scratch, allowing users to quickly create programs that perform visible actions. Yet, they can result in the user forming bad programming habits because proper software engineering practices (e.g. code reuse) are not formally introduced (Dorling & White, 2015). This is particularly important because of the widespread use of Scratch in primary education and the lack of teacher CS expertise, meaning that children often use Scratch without structured lessons or adequate support. Other tools that are similar to Scratch, like Hopscotch and Tynker, also have these problems.

Bad Programming Habits

Meerbaum-Salant, Armoni & Ben-Ari (2011) found that children age 14 and 15 demonstrated two ‘habits of programming’ that are at odds with accepted CS and software engineering practices. The first of these is extreme bottom-up programming. When correctly used, a bottom-up approach enables the programmer to design and develop components separately before they are integrated to form a top-level system. However, it was observed that instead of thinking about tasks from an algorithmic level, children would drag all the blocks into the program that seemed appropriate for solving the task, only then combining them to form scripts (Figure 5.1). This pattern of behaviour is characterised as programming by ‘bricolage’ and is closely tied with constructionism. An example of bricolage is a chef who constructs a dish as he goes along (bottom-up) instead of following a recipe (top-down). There are suggestions that programming by ‘bricolage’ can favour higher-ability children and that those of lower-ability need more structured support (Rose, 2016).

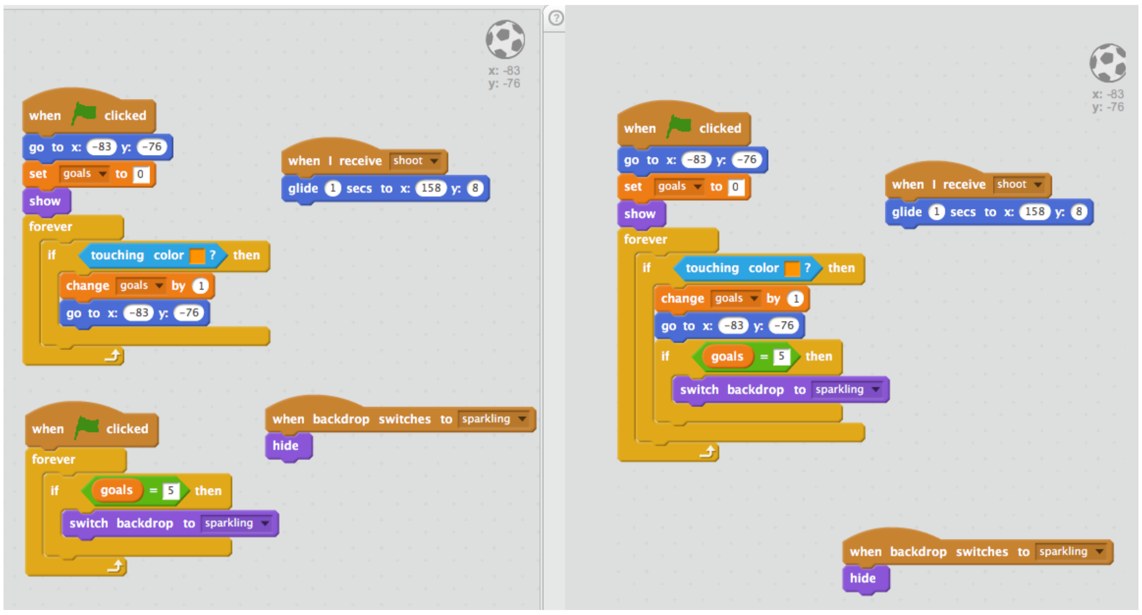


```
when [flag] clicked
switch backdrop to goal1
go to x: -160 y: -52
forever
    if key space pressed? then
        move 10 steps

switch costume to costume2

turn 15 degrees

when space key pressed
broadcast shoot

change x by 10

forever
    if then

define shoot

x position
```

Figure 5.1: Screenshot of an example of bottom-up or ‘bricolage’ programming, where blocks are dragged into the program that seem appropriate for the task, without much thought put into the program design

The second habit is ‘extremely fine-grained programming’ (EFGP), which complements the first habit by taking the top-down programming approach to its extreme. Top-down is the traditional software engineering method for writing programs, breaking down (or decomposing) a problem into a modular structure that forms a complete solution. The authors found that children would decompose scripts until they became extremely small, often lacking logical coherency (Figure 5.2). Some programs comprised of hundreds of these scripts, making them difficult to understand because Scratch executes all scripts concurrently. This concurrency can be useful if consciously designed but otherwise makes programs difficult to debug. It is worth noting that both these problems were observed whilst children followed a textbook that emphasised program analysis and design.

The extreme bottom-up and top-down approaches observed by Meerbaum-Salant, Armoni & Ben-Ari (2011) suggest that Scratch encourages certain programming habits that lead to programs that are difficult to understand, debug and maintain. If left unchecked, these habits may influence the learner as they switch to text-based languages (Weintrop & Wilensky, 2015). These bad programming habits can be identified and addressed if the programmer can ‘smell’ that something is wrong with their code and can ‘refactor’ it to remove the issues. Yet, this skill is rarely taught before higher education.



```
x: -83
y: -76

when clicked
go to x: -83 y: -76
set goals to 0
show
forever
    if touching color ? then
        change goals by 1
        go to x: -83 y: -76

when I receive shoot
glide 1 secs to x: 158 y: 8

when clicked
forever
    if goals = 5 then
        switch backdrop to sparkling

when backdrop switches to sparkling
hide
```

```
x: -83
y: -76

when clicked
go to x: -83 y: -76
set goals to 0
show
forever
    if touching color ? then
        change goals by 1
        go to x: -83 y: -76
    if goals = 5 then
        switch backdrop to sparkling

when I receive shoot
glide 1 secs to x: 158 y: 8

when backdrop switches to sparkling
hide
```

Figure 5.2: Screenshot of an example of extreme decomposition or EFGP (left), the two ‘when green flag clicked’ scripts have been decomposed and would be more logically coherent if combined (right)

5.2.2 Code Smells in Object-Oriented Programming

The term ‘code smell’ was coined by Fowler (1999) in his book ‘Refactoring: Improving the Design of Existing Code’. A code smell is a surface indication in a program that usually corresponds to a deeper problem. Code smells can help the programmer identify parts of their code that need ‘refactoring’, that is “the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure” (p. 9). The refactoring process involves working at multiple levels of abstraction because the programmer must be able to understand the code at the Program (2), Object (3) and Problem (4) levels of the PGK hierarchy.

Fowler gives a long list of possible code smells that includes duplicated code, long methods, large classes and long parameter lists, explaining in each case the refactoring methods that can be used to remove them. Keuning, Heeren & Jeuring (2017) found that university students rarely fix code quality issues, in particular, issues related to modularisation or system design, even when given code analysis tools.

5.2.3 Code Smells in Scratch

Several exploratory studies of large repositories of Scratch projects have shown that duplicated code, large scripts and dead code smells are common (Table 5.1). Hermans & Aivalaglou (2016) found that these smells can impact understanding, debugging and the ease with which learners can alter projects. Furthermore, Techapalokul & Tilevich (2015) found that novice programmers “prone to introducing some smells continue to do so even as they gain experience” (p. 10). Code smells are particularly important because ‘remixing’ other users’ projects is a large part of the Scratch online platform (Dasgupta et al., 2016).

Table 5.1: Percentage of code smells and the use of procedures in analyses of Scratch projects

Author(s)	Projects Analysed	% of Projects			
		Duplicated Code	Dead Code	Long Scripts	Procedures
Moreno-León & Robles (2014)	100	62%			17%
Aivalaglou & Hermans (2016)	247,798	26%	28%	30%	8%
Techapalokul (2017)	1,066,308	46%	23%	47%	
Robles et al. (2017)	250,166	20%			8%

In another work, Techapalokul (2017) listed 12 different code smells that appear in Scratch (Table 5.2). This chapter focuses on duplicated code, long scripts and dead code (unused custom blocks, unused variables and unreachable code) as these have been included in other analyses. The rest of this section examines these code smells in Scratch, exploring the frequency that they appear and the problems that they cause. The next section explains how duplicated code and long script smells can be removed through procedural abstraction and code reuse.

Duplicated Blocks

Moreno-León & Robles (2014) analysed 100 randomly selected Scratch projects from the Scratch online repository. They were analysed using Hairball (Boe et al., 2013), a static code analyser for Scratch written in Python. The authors wanted to detect two bad programming habits that they had observed in high school students: default sprite names and duplicated code. Hairball works by analysing the 'tokens' of the blocks in a Scratch program. These are textual representations with generalised inputs, for example 'move %s steps'. This means that two blocks are considered equal if it is only their input values that differ. Duplicated code was classified as a minimum of five blocks where the tokens match. They found that 62% of projects contained duplication.

A larger analysis of 247,798 projects by Aivalaglou & Hermans (2016) used the same five-block threshold for script duplication. They found that 26% of projects had scripts duplicated across sprites, 10% had scripts duplicated within sprites and 11% of projects contained exact duplication (with input values the same). Techapalokul (2017) found that 46% of projects contained duplicated code in an analysis of 1,066,308 projects. However, Techapalokul does not specify the duplicated block threshold used.

Table 5.2: Code smells in Scratch (Techapalokul, 2017, p. 782)

Name	Description
Broad Variable Scope	A variable with its scope broader than its usage does not tell which scriptable the variable belongs. Too many global variables clutter script palette and drop-down menus.
Duplicate Code	Repeated sequence of blocks regardless of block arguments is used as a way to reuse code.
Duplicate String	Same string values are repeatedly used in multiple program locations.
Extreme Fine-Grained Script	Breaking up of functionally similar scripts into several small fine-grained event-based scripts.
Hard-Coded Media Sequence	A sequence of media elements is hard-coded as block arguments.
Long Script	Long script (longer than 11 blocks) suggest inadequate decomposition and hinder code readability.
Uncommunicative Naming	Generic naming started with Sprite or message (e.g. “Sprite2” and “message1”) make programs unreadable.
Unnecessary Workaround	Use of polling of flag variables to direct control flow to recreate broadcast-receive mechanism.
Unorganised Script	Similar event-based scripts are scattered around making the program hard to navigate.
Unreachable Code	An unreachable script can be safely removed without affecting the program behaviour.
Unused Custom Block	A script definition of an unused custom block can be safely removed without affecting the program behaviour.
Unused Variable	A variable is declared but never used anywhere in the program.

Hermans & Aivalaglou (2016) found that duplication, whether it be blocks or scripts, makes Scratch projects more difficult to modify and maintain. Students performed significantly worse on Scratch code comprehension tasks in projects that had long script and duplication smells. Block duplication indicates a lack of abstraction and decomposition skills and violates the DRY (Don’t Repeat Yourself) software engineering principle.

Long Scripts

In addition to analysing for script duplication, Aivalaglou & Hermans (2016) also analysed Scratch projects for long scripts. They classified long scripts using the top 10% largest scripts in the 247,798 projects they analysed, setting the threshold for a large script at 18 blocks. They found that 30% of projects contained scripts of 18 blocks or more. Techapalokul (2017) used a slightly lower threshold of 11 blocks, finding that 47% of the 1,066,308 projects analysed contained a long script. Long scripts suggest inadequate decomposition and can hinder code readability.

Dead Blocks

Aivalaglou & Hermans (2016) also analysed Scratch projects for dead code: blocks that are either not invoked (not attached to an event block), empty event scripts (the script block alone), procedures that are not invoked and unmatched broadcast-receive messages. They found that 28% of projects contained one or more of these dead code smells. Techapalokul (2017) used similar criteria for dead code: unused custom blocks, unused variables and unreachable code (an unreachable block can be safely removed without affecting the program behaviour). Finding that 29% of projects contained unused custom blocks, 25% contained unused variables and 23% contained unreachable code.

Dead blocks are an indicator of the bottom-up or 'bricolage' programming approach described in Section 5.2.1. Meerbaum-Salant, Armoni & Ben-Ari (2011) found that children would drag all the blocks into the program that seemed appropriate for solving the task, only then attempting to compose a solution. Inevitably, some of these blocks are then not removed. Scratch encourages this because it does not indicate that these scripts will not be executed (dead blocks are 'greyed out' in similar tools like Google Blockly and LEGO Mindstorms EV3.) Dead code can be distracting and confusing to the programmer and in the case of unused variables and unused custom blocks, indicate that inadequate thought has been put into program design.

5.2.4 Code Smells in Other Educational Programming Tools

Hermans, Stolee & Hoepelman (2016) analysed code smells in two other educational programming tools, Lego Mindstorms EV3 and Kodu Game Lab. They analysed projects from both for 11 different object-oriented inspired code smells, including dead code, message chaining (multiple calls or jumps between different objects) and unused variables. In an analysis of 44 projects, they found that 88% of EV3 projects and 93% of Kodu projects contained at least one smell. The most common smells (appearing in at least a third of programs) were lazy classes (objects with few blocks), duplicated code and dead code.

GameMaker's drag and drop language goes some way to limiting code duplication and dead code. Objects have 'events' that cannot be duplicated, unlike in Scratch where there can be many of the same event block (e.g. when something is clicked) in one sprite. However, 'actions' in each event can be applied to another object (Figure 5.3), meaning that duplication can exist where the same action (or sequence of actions) is applied to an object in different places. For example, collision events can be added to both objects in the collision, leading to potential logic duplication. Dead code is rarer in GameMaker projects because actions must be chained with other actions.

In summary, code smells are possible in all programming tools that contain enough complexity. Duplicated code and long methods, in particular, can be found in any pro-

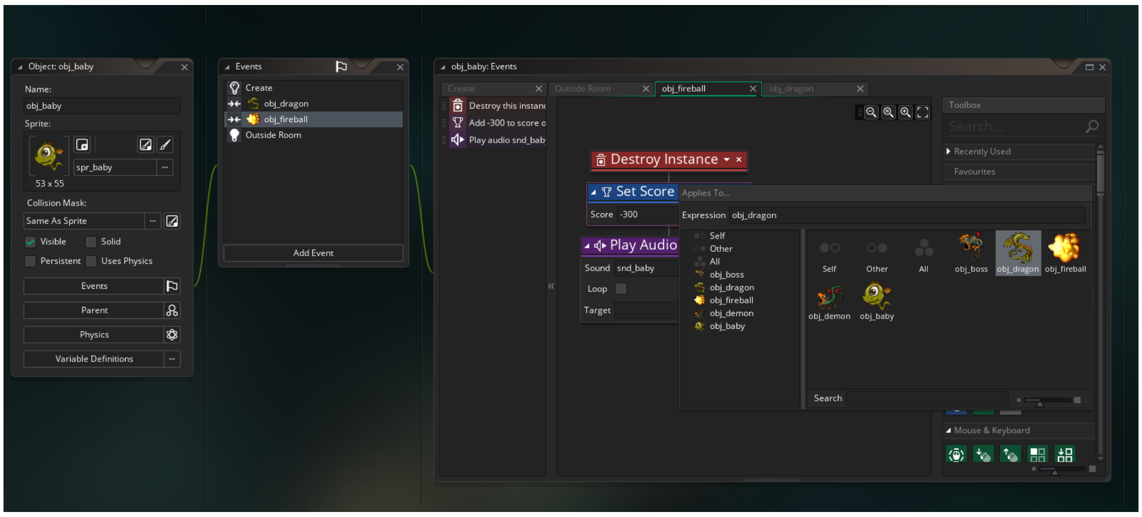


Figure 5.3: Screenshot of objects, events and actions in GameMaker Studio 2

gramming environment. Whereas dead code is often a by-product of block-based tools where blocks can exist in the program without being connected to anything else. Yet, it could apply to code that is 'unreachable' behind a conditional statement that is never true, which is possible in all programming tools that contain conditional execution.

5.3 Dealing with Code Smells

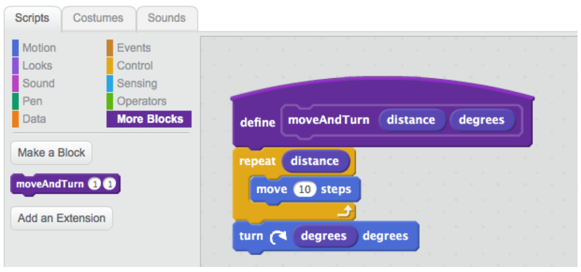
Section 5.2.3 demonstrated that duplicated code, long scripts and dead code are all common in Scratch projects. These smells can impact understanding, debugging and project maintenance. To remove code smells, novice programmers must be able to recognise them and alter their code to remove them, whilst maintaining the behaviour of the program. The solution to dead code is simply to remove it, yet duplication and large scripts are more complicated and require some restructuring. Fowler (1999) recommends the 'extract method' for dealing with duplicated code and large methods in object-oriented programming. This method can also be used in Scratch.

5.3.1 The Extract Method

The extract method is recommended by Fowler (1999) to deal with duplicated code and long methods. It involves moving fragments of code into a procedure with or without parameters (to pass variable information) that can then be invoked from multiple places. These methods should be small and well-named.

From version 2.0, Scratch enables procedures by allowing users to create blocks, which can then be used (or invoked) in multiple places within the sprite. Information can be passed to these 'custom blocks' through number, text and boolean parameters. The user can refactor code using the extract method by creating a custom block to encapsulate a set of repeated commands (Techapalokul & Tilevich, 2015). This is the example of abstraction in CS used by Barr & Stephenson (2011) in their CT definition.

Dr. Scratch (Section 3.3.1) gives Scratch projects a score out of 21 across seven CT concepts based on the blocks used. It uses custom blocks (2 points) and clones (3 points) as a measure of abstraction in projects, combining this with decomposition (having multiple scripts in multiple sprites, 1 point). Table 5.1 shows that this functionality is not often used and explains the common appearance of duplicated code and long scripts in Scratch programs. The next two sections will explain Scratch's custom blocks and clones in more detail.



```
define moveAndTurn (distance) (degrees)
repeat (distance)
    move (10) steps
turn ↻ (degrees) degrees
```

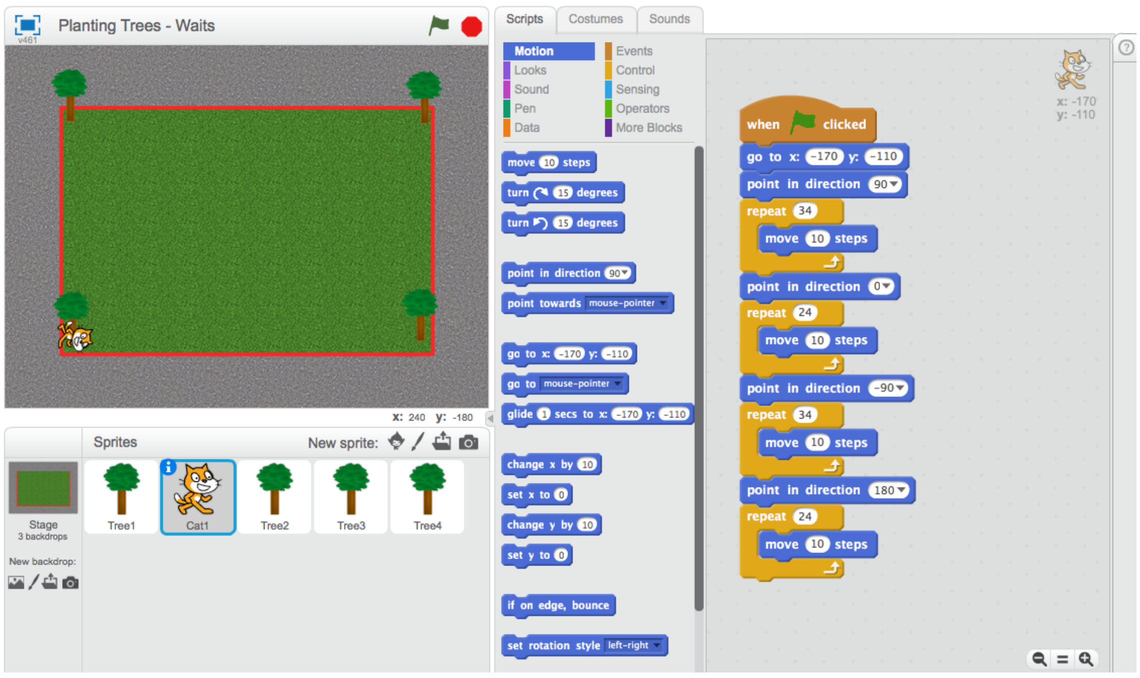
```
void moveAndTurn(distance, degrees) {
    for (int i = 0; i < distance; i++) {
        moveSteps(10);
    }
    turnRight(degrees);
}
```

Figure 5.4: Screenshot of a custom block that moves and turns a sprite, taking arguments for distance and direction (left) and its implementation as a procedure in C++ (right) for comparison

5.3.2 Custom Blocks

'More Blocks' (or 'My Blocks' in Scratch 3.0) are a category of blocks that hold custom procedures for the selected sprite (far left of Figure 5.4). The category starts off empty apart from a 'Make a Block' button, which, when clicked, brings up a dialogue box for creating a block. This allows the user to specify the name of the block and any parameters (known as 'inputs' in Scratch) that they want the block to have. When created, a 'define' block is added to the program, letting the user specify what will happen when the block is executed. The block appears in the 'More Blocks' category and can be used like any other Scratch block. Figure 5.4 shows an example of a custom block in Scratch and its corresponding implementation in C++. Custom blocks in Scratch are limited in that they cannot be used return values like functions in text-based programming languages and can only be defined for the current sprite. Snap! (Harvey, Garcia, Paley, & Segars, 2012) is a Scratch-based programming tool that enables full procedures but is aimed at children age 12 and above.

Figure 5.5 shows an example of a Scratch project that contains a duplicated code smell: point in direction, repeat and move are all repeated four times with different input values. This smell has been refactored in Figure 5.6 using the extract method. The duplicated code has been moved into a procedure called 'turnAndMove' that takes two arguments, distance and degrees. The Scratch user has recognised the duplicated code smell and extracted that functionality into a reusable procedure, noticing and



```
when [flag] clicked
go to x: -170 y: -110
point in direction 90
repeat 34
    move 10 steps
point in direction 0
repeat 24
    move 10 steps
point in direction -90
repeat 34
    move 10 steps
point in direction 180
repeat 24
    move 10 steps
```

Figure 5.5: Screenshot of a Scratch project that uses block duplication to move a cat around the four corners of a park and plants a tree on each corner

including arguments for the values that change. This process is known as ‘procedural abstraction’, which Kallia & Sentance (2017) describe as ‘threshold concept’ in CS and an area that is difficult for students at high-school and university level to understand. The notion of a threshold concept is described by Meyer & Land (2003) as “opening up a new and previously inaccessible way of thinking about something. It represents a transformed way of understanding, or interpreting, or viewing something without which the learner cannot progress” (p. 1). Procedural abstraction is, therefore, an important skill for novice programmers to learn. Despite this, procedures (custom blocks) are not often taught or used in Scratch projects.

This chapter has already discussed the development of abstraction skills as a programmer becomes more experienced. Despite the difficulties that novices have with procedural abstraction, it should be possible to introduce custom blocks and the concept of code reuse to primary school children if they are given concrete examples of where and when they should be used (Armoni, 2012).

5.3.3 Cloning

Cloning is a feature in Scratch that allows a sprite to create a clone of itself (or another sprite) while the project is running. This duplicate is a separate instance of the original or parent sprite but will inherit scripts, costumes, sounds and properties that can be modified. Cloning a sprite is similar to creating an instance of a class in object-oriented programming, where an ‘instance’ is a concrete occurrence of an object that is created during runtime. There are three blocks in Scratch related to cloning: ‘create

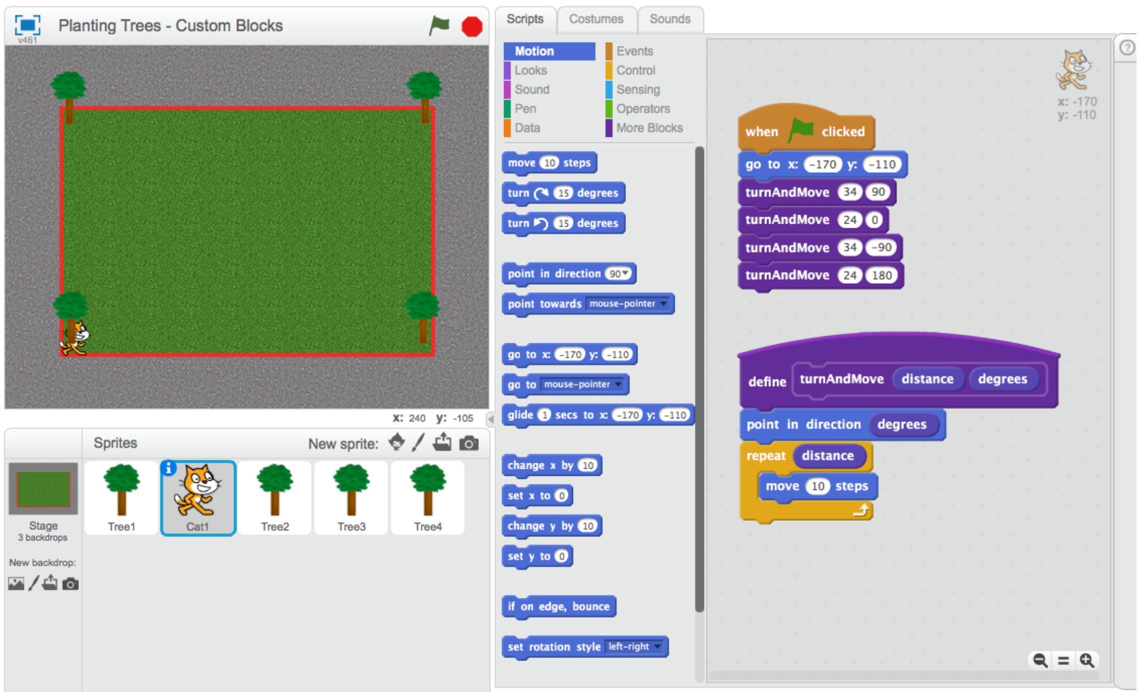
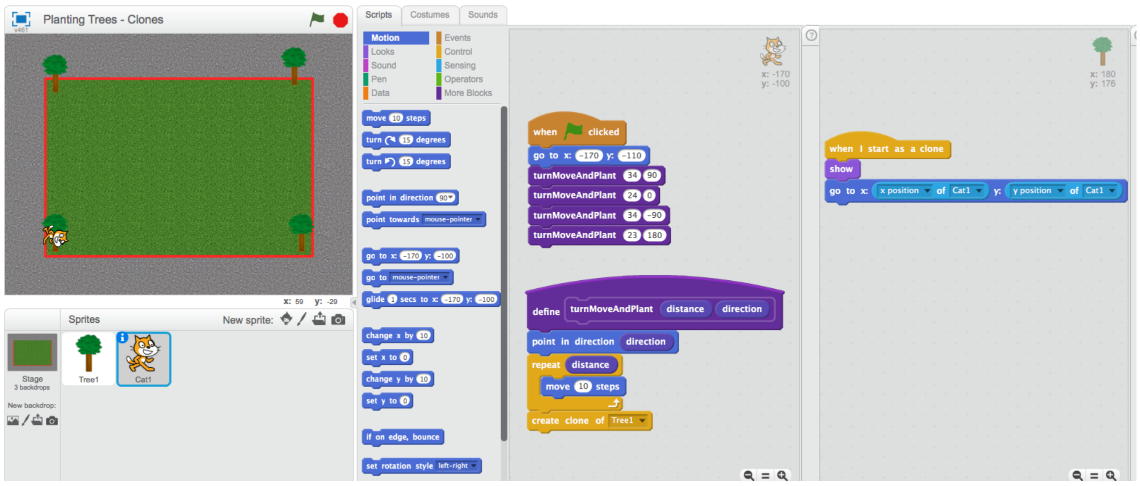

```
when [flag] clicked
go to x: -170 y: -110
turnAndMove 34 90
turnAndMove 24 0
turnAndMove 34 -90
turnAndMove 24 180

define turnAndMove (distance) (degrees)
point in direction (degrees)
repeat (distance)
    move 10 steps
```

Figure 5.6: Screenshot of the Scratch project in Figure 5.5 where the extract method has been used to create a custom block that reduces code duplication

clone of', 'when I start as a clone' and 'delete this clone'. Cloning is an important part of games and projects that require more than one instance of a sprite, for example, special effects like fireworks or snow. Without clones, scripts would be duplicated across almost identical sprites (see Figures 5.5 and 5.6). This violates the DRY (Don't Repeat Yourself) principle and produces duplicated code smells. Cloning can be used to reduce this duplication. Figure 5.7 shows a further refactor of Figure 5.6 that uses tree clones to reduce the number of sprites, therefore removing the duplicated code across them.


```
when [flag] clicked
go to x: -170 y: -110
turnMoveAndPlant 34 90
turnMoveAndPlant 24 0
turnMoveAndPlant 34 -90
turnMoveAndPlant 23 180

define turnMoveAndPlant (distance) (direction)
point in direction (direction)
repeat (distance)
    move 10 steps
create clone of Tree1
```

```
when I start as a clone
show
go to x: (x position of Cat1) y: (y position of Cat1)
```

Figure 5.7: Screenshot of the Scratch project in Figure 5.6 but using tree clones (right) to reduce the number of sprites

Design Concept - Similarity to Scratch
Creating a programming game that teaches children to use abstraction to reduce code smells in Scratch needs to be similar enough to it that the skills can transfer.

5.3.4 Custom Blocks and Cloning in Practice

Robles et al. (2017) measured 250,166 projects using Dr. Scratch whilst also analysing them for duplicated code smells. They found that the use of custom blocks (8% of projects) and cloning (10%) did not impact the amount of code duplication, even for high-scoring projects. In other words, custom blocks and clones were not used to refactor code. This may be because most Scratch manuals do not sufficiently address code duplication and how to solve it using custom blocks. In addition, the constructionist nature of Scratch with no error messages or explanations means that this functionality is too difficult for a novice to understand without prior instruction. In primary education, where teachers are often not trained in CS, it is unlikely they will be able to effectively demonstrate these abstraction techniques to children.

5.4 Summary

In summary, abstraction is an essential skill in CS and CT. In CS, the main objective of abstraction is the process of 'information hiding' or hiding details that are essential in one context but inessential in another. Abstraction is also important in software engineering, it is used to model components of a problem domain and to reduce duplication, ensuring that the programmer does not violate the DRY (Don't Repeat Yourself) software engineering principle. Good computer scientists can move quickly and efficiently between multiple levels of abstraction, seeing things both 'in the large' and 'in the small'.

Novice programmers develop abstraction skills as they gain experience. They begin by only being able to trace code using specific values, eventually progressing to being able to understand chunks of code simply by reading. The neo-Piagetian view is that these skills can be developed regardless of age, as long as working memory is sufficient. This indicates that it should be possible to develop abstraction skills in older primary school children (between age 9 and 11).

Scratch is the most popular programming tool in primary education. Its design encourages a constructionist, self-directed learning approach where solutions are created through exploration. However, this can result in bad programming habits such as extreme bottom-up and top-down approaches, where programs lack formal design and logical coherency. These problems can be addressed if a programmer is taught to 'smell' that something is wrong with their code. A code smell is a surface indication in a program that usually corresponds to a deeper problem. Code smells indicate that a program should be refactored, where the internal structure of the code is improved without altering its external behaviour.

Duplicated blocks and sprites, long scripts and dead code are all common code smells in Scratch and make projects difficult to understand, debug and maintain. The first two can be refactored using the extract method, which involves turning a section of code into a procedure that can then be invoked in multiple places. In Scratch, procedures take the form of custom blocks. These blocks are created by the user and can be passed data through number, text and boolean arguments. Using procedures correctly requires procedural abstraction, which is a threshold concept in CS. Duplication smells can also be removed in Scratch using cloning, where a sprite creates a clone of itself (or other sprites) at runtime. Without cloning, sprites are often copied and pasted, resulting in both block and sprite duplication.

Despite custom blocks and clones, Scratch users still frequently copy and paste code in their projects. This is because these concepts are difficult to understand without formal teaching, particularly for children. Most Scratch manuals do not sufficiently address code duplication and how to reuse code. Furthermore, weaknesses in teacher CS expertise, particularly in primary education, mean that teachers are unable to effectively teach children how to use abstraction.

The importance of abstraction in CS and CT means that children should be developing these skills as soon as they are able. Custom blocks and clones in Scratch give concrete examples of abstraction in an accessible environment. The next chapter (Chapter 6) will explore whether primary school children age 10 and 11, with some Scratch experience, can recognise the benefits of abstraction.

Chapter 6

Study 2 - Can Children Recognise the Benefits of Abstraction in Scratch?

Chapter 5 examined the ‘code smells’ common in Scratch projects: duplicated code, long scripts and dead code. These smells can make projects difficult to understand, debug and maintain (Hermans & Aivaloglou, 2016). A good programmer can identify these smells and ‘refactor’ their code to remove them (M. Fowler, 1999). Refactoring is the process of improving the internal structure of a system without altering its external behaviour and requires the programmer to work at multiple levels of abstraction.

Duplicated code and long scripts can be refactored using the extract method: moving fragments of code into a procedure that can be invoked from multiple places. Scratch enables procedures through custom blocks. Using procedures to reduce code duplication requires procedural abstraction, which is a vital skill in computer science (CS). In Scratch, sprite duplication can be removed through cloning (creating instances of a sprite). The Dr. Scratch computational thinking (CT) assessment uses custom blocks and cloning to measure abstraction in Scratch projects. However, research has shown that these are not often used in projects, and when they are, they do not impact the amount of duplication in projects (Robles et al., 2017). Using custom blocks and clones requires good abstraction skills, which primary school children should be able to learn if the content is presented in a structured way (Gibson, 2012).

This chapter describes a formative study to see if children age 10 and 11 with limited Scratch experience can recognise the benefits of abstraction in Scratch using custom blocks and clones. This was done to establish the potential for teaching abstraction to this age group, before focusing on the development of a game to support this understanding that is described in the next chapter (Chapter 7).

The chapter explains the background of the study, before moving onto the method, results, discussion and conclusion.

6.1 Background

6.1.1 Abstraction

Chapter 5 discusses abstraction as a general aspect of human cognition, describing the conceptual process where general rules are derived from concrete examples. In CT, it is the process of ‘removing the detail from a problem and formulating solutions in generic terms’ (Table 3.4). In CS and software engineering, abstraction is used to hide information between the levels of a program, remove duplicated logic through procedural abstraction and to model the problem domain. Good computer scientists can move easily between multiple levels of abstraction (Armoni, 2016).

Novice programmers develop abstract reasoning skills as they gain expertise (Lister, 2011). There are indications that primary school children can learn abstract CS concepts through concrete objects and physical manipulation of information (Gibson, 2012). However, the observations of Benton et al. (2018) suggest that children under age 11 can struggle to compare different algorithms, particularly those that contain procedural abstraction. Swidan, Hermans & Smit (2018) found that children can hold misconceptions about programming concepts like variables, loops and conditionals. This indicates that structured teaching is required for children to learnt to use abstraction correctly.

6.1.2 Code Smells

A code smell is a surface indication of an underlying problem in a program (M. Fowler, 1999). Code smells indicate that a project needs to be ‘refactored’, where the internal structure of a system is altered without changing its external behaviour. Possible smells in object-oriented programming include duplicated code, long methods, large classes and long parameter lists.

Chapter 5 discusses several studies of large Scratch repositories that showed that duplicated code, long scripts and dead code are all common code smells in Scratch projects. Code smells can indicate bad programming habits that are at odds with accepted software engineering practices (Meerbaum-Salant et al., 2011). These problems can be exacerbated by Scratch’s self-directed, constructionist design where the block palette is visible at all times, there is little in-built feedback and there are no error messages. Hermans & Aivaloglou (2016) found that code smells hampered novice programmers, making projects difficult to understand, debug and maintain. This is problematic because Scratch’s online platform is built on the ability to ‘remix’ other users’ projects (Dasgupta et al., 2016). Moreover, programming habits picked up in block-based tools may influence the learner as they transition to text-based languages (Weintrop & Wilensky, 2015).

Therefore, novice programmers should be taught to ‘smell’ that something is wrong with their code and are then able to ‘refactor’ it to remove the smell. In Scratch, duplicated code and long script smells can be removed by reusing code through custom blocks (procedures) and cloning (creating instances of a sprite).

6.1.3 Custom Blocks and Cloning

Custom blocks (or ‘More Blocks’/‘My Blocks’) are a category of blocks in Scratch that hold procedures for the selected sprite (Section 5.3.2). They can be used to reuse code through the extract method, where fragments of code are moved into a procedure (or custom block) that can then be used in multiple places. This process requires procedural abstraction, which Kallia & Sentance (2017) suggest is a threshold concept in CS: opening a door to a new way of thinking about programming. Cloning in Scratch allows a sprite to create a clone of itself or another sprite while the project is running (Section 5.3.3). This can be used to duplicate sprites (e.g. fireworks or clouds) that have the same behaviour. Cloned sprites inherit scripts, costumes, sounds and properties from the original sprite.

Dr. Scratch (Section 3.3.1) measures Scratch projects for CT, giving them a score of 21 across seven concepts based on the blocks used. It measures abstraction using custom blocks and cloning. Giving the user 2 points for using custom blocks and 3 points for cloning. However, this functionality is rarely used in Scratch projects (Table 5.1). Dr. Scratch combines abstraction with decomposition, giving 1 point for having multiple scripts in multiple sprites.

Custom blocks and clones enable the user to reuse code, therefore avoiding the copying and pasting duplication that violates the DRY (Don’t Repeat Yourself) software engineering principle (Hunt & Thomas, 1999). This makes projects easier to understand, debug and maintain. Yet, even when they are used, Scratch users still frequently copy and paste code (Robles et al., 2017), indicating that they are not used correctly. This may be because code reuse and refactoring require abstract reasoning skills, which are only developed as the programmer gains expertise. Code reuse is difficult to understand without formal teaching, which is problematic at primary school level because teachers often lack technical knowledge and confidence in delivering CS (Chapter 2). However, there are indications that primary school children (under age 11) can understand abstract CS concepts if learning content is presented in a structured way (Gibson, 2012).

6.2 Method

This chapter describes a formative study designed to see whether can children age 10 and 11 with limited Scratch experience can recognise the benefits of abstraction. Groups of children were asked to rank four Scratch projects with the same external behaviour, each project achieved a different score for abstraction and decomposition in Dr. Scratch. The rankings were based on several criteria: ‘best coded’, easiest to understand, smallest scripts and least blocks and easiest to change.

6.2.1 Participants

The study participants were 21 children age 10 and 11 from a medium-sized primary school in northern England. The school is around the national average for pupils meeting the expected standard in reading, writing and maths (63%). All participants had a term of Scratch teaching in the previous academic year, but no experience using custom blocks or cloning.

6.2.2 Materials

The participants were asked to rank four Scratch projects produced to the same specification: *‘when the green flag is pressed, animate the cat’s movement around the edge of the park, planting a tree in each corner’*. The projects demonstrate the four levels of abstraction and decomposition measured by Dr. Scratch. They all start when the ‘green flag’ button is clicked and have a ‘go to position’ block that resets the position of the cat sprite to the bottom left corner of the park. Each project was given a name to suggest to participants that it was produced by a child in another school. Note, three of the projects were shown in the last chapter (Chapter 5) as examples of custom blocks and cloning (Figures 5.5, 5.6 and 5.7). Table 6.1 shows a summary of each project, the rest of this section then describes them in more detail.

Ava

The first project gets 0 points for abstraction and decomposition in Dr. Scratch (Figure 6.1). The cat moves around the four corners of the park using several repeated ‘point in direction’, ‘repeat’ and ‘move’ blocks (a block duplication smell). At 14 blocks, this script would be classified by Techapalokul (2017) as a long script smell. The trees are copied and pasted (sprite duplication) and do not contain any scripts, which means that they are not ‘planted’ as specified. The project does not strictly meet the specification and has a slightly different outcome to the others.

Emma

The second project is similar to the first. Yet, it gets 1 point for abstraction and decomposition because it has multiple scripts in multiple sprites, showing evidence of decomposition. The script in the cat is identical (block duplication and long script smells). However, each tree (sprite duplication) is hidden and shown after the number of seconds that it takes the cat to reach that corner of the park when the project is run (Figure 6.2).

Table 6.1: Scratch projects breakdown

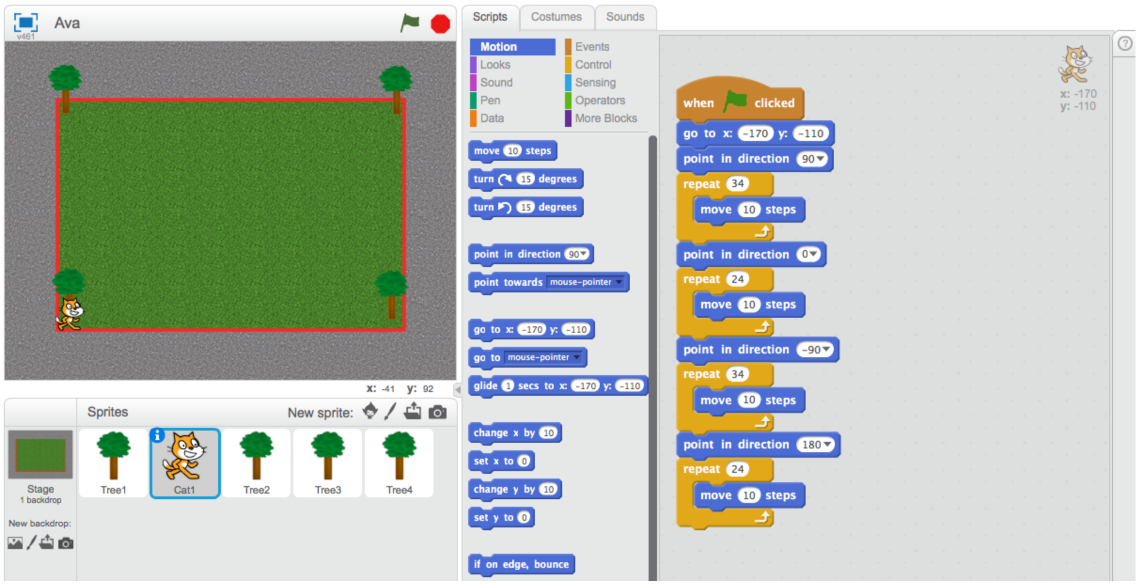
Project Identifier	Dr. Scratch Abstraction and Decomposition Score	Sprites (Number of Blocks)	Description	Code Smell(s)
Ava	0	Cat (14) Tree1 (0) Tree2 (0) Tree3 (0) Tree4 (0)	The cat moves around the four corners of the park but the trees are already in place (as these sprites are empty.)	Block and sprite duplication, long script (14 blocks)
Emma	1	Cat (14) Tree1 (4) Tree2 (4) Tree3 (4) Tree4 (4)	The cat moves around the four corners of the park, the trees are on timers to 'wait' for the cat to reach each corner.	Block and sprite duplication, long script (14 blocks)
Alice	2	Cat (10) Tree1 (4) Tree2 (4) Tree3 (4) Tree4 (4)	The cat moves around the four corners of the park (using a custom block), the trees are on timers to 'wait' for the cat to reach each corner.	Sprite duplication
Zoe	3	Cat (11) Tree (3)	The cat moves around the four corners of the park (using a custom block), a clone of the tree sprite is created at the position of the cat when it reaches each corner.	None

Alice

The third project uses a custom block (Figure 6.3) to remove the block duplication smell in the cat, achieving 2 points for abstraction and decomposition in Dr. Scratch. The blocks for turning and moving along each edge of the park, 'point in direction', 'repeat' and 'move', have been extracted into a custom block called 'turnAndMove' that takes two parameters, degrees and distance. Degrees specifies the direction to point the cat in and distance is the number of times the step block will be repeated. The block is then used four times, one for each side of the park. The trees are still duplicated sprites that are hidden and shown after several seconds.

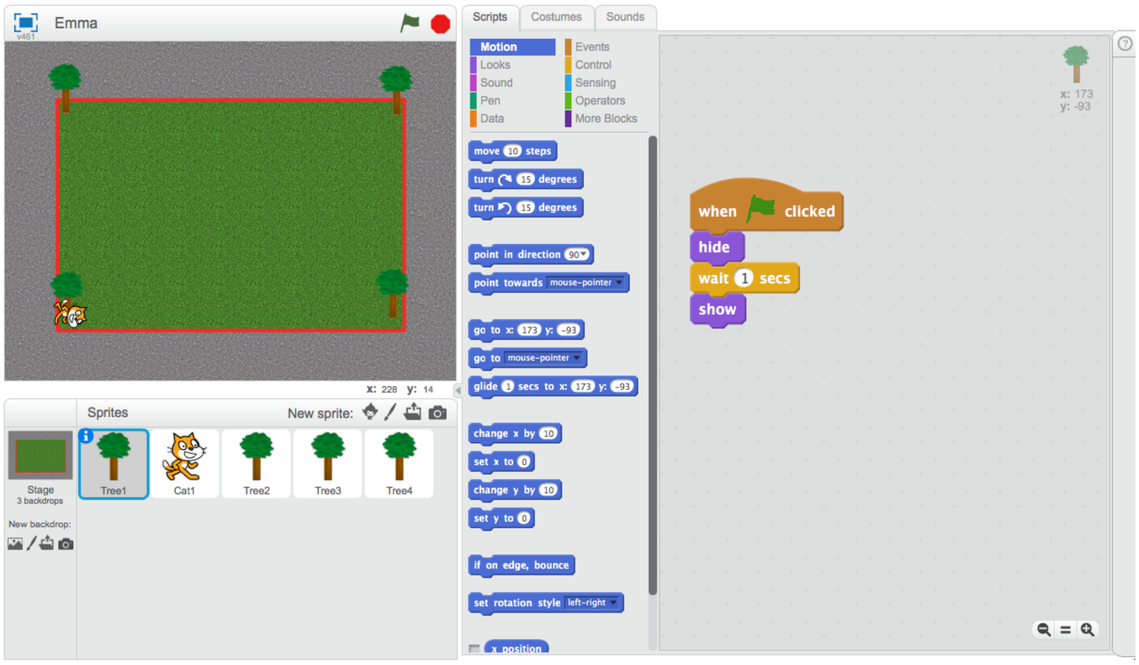
Zoe

The fourth project gets 3 points for abstraction and decomposition in Dr. Scratch. In addition to the 'turnAndMove' custom block (renamed 'turnMoveAndPlant'), the fourth project uses cloning to 'plant' a single tree sprite on each corner (Figure 6.4). When the cat reaches a corner, a clone of the tree sprite is created in that position (removing the sprite duplication).



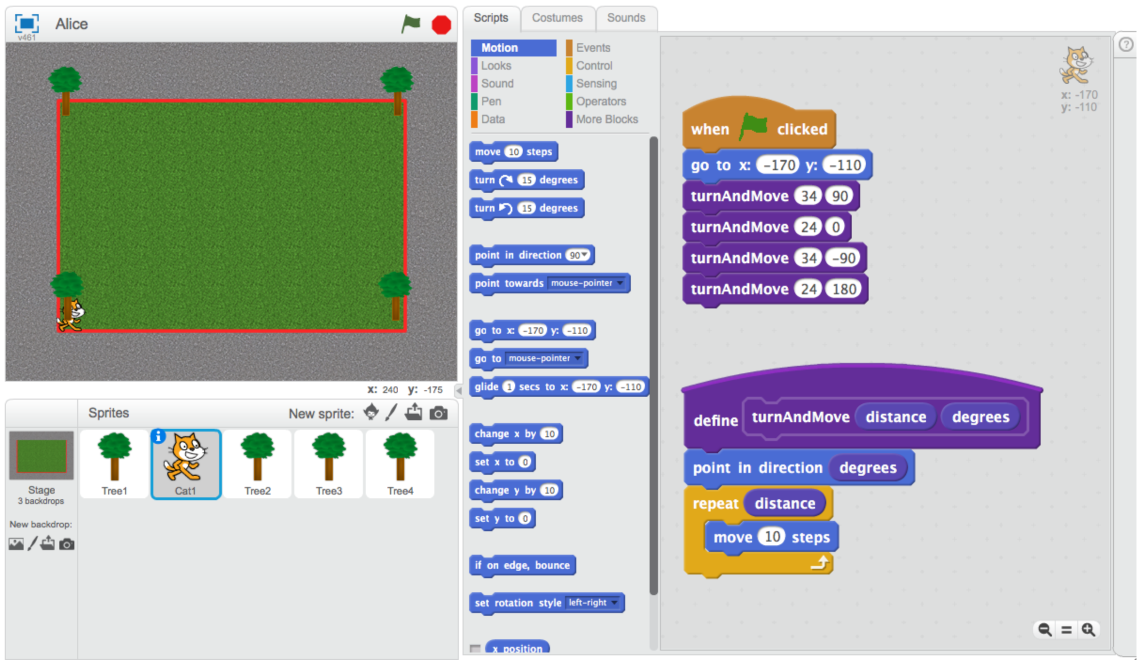
```
when 🏴 clicked
go to x: -170 y: -110
point in direction 90
repeat 34
    move 10 steps
point in direction 0
repeat 24
    move 10 steps
point in direction -90
repeat 34
    move 10 steps
point in direction 180
repeat 24
    move 10 steps
```

Figure 6.1: Screenshot of Ava's Scratch project that gets 0 points for abstraction and decomposition in Dr. Scratch



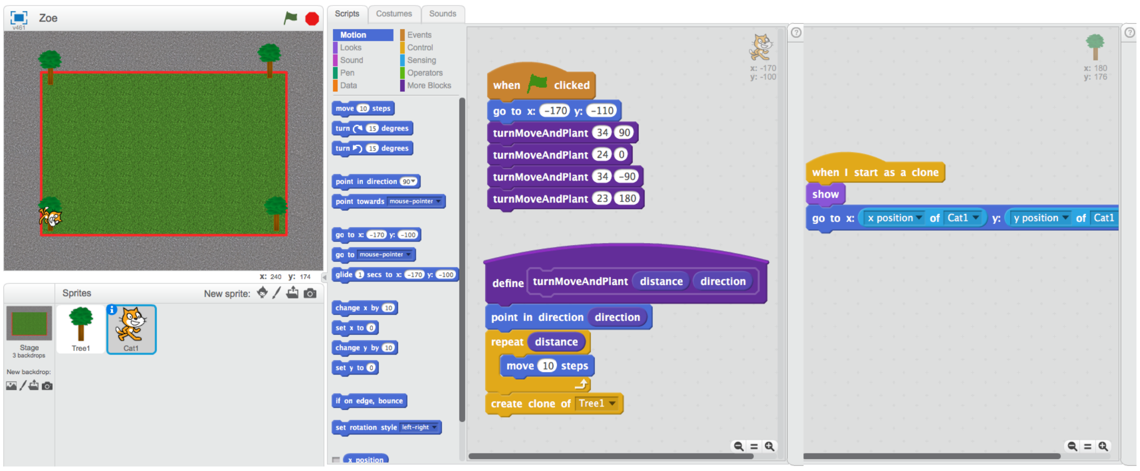
```
when 🏴 clicked
hide
wait 1 secs
show
```

Figure 6.2: Screenshot of Emma's Scratch project that gets 1 point for abstraction and decomposition in Dr. Scratch (each tree has a different number of seconds in the 'wait' input and the cat sprite the same as in Figure 6.1)



```
when [flag] clicked
go to x: -170 y: -110
turnAndMove 34 90
turnAndMove 24 0
turnAndMove 34 -90
turnAndMove 24 180

define turnAndMove distance degrees
point in direction degrees
repeat distance
    move 10 steps
```

Figure 6.3: Screenshot of Alice's Scratch project that gets 2 points for abstraction and decomposition in Dr. Scratch (tree sprites are the same as they are in Figure 6.2)



```
when [flag] clicked
go to x: -170 y: -110
turnMoveAndPlant 34 90
turnMoveAndPlant 24 0
turnMoveAndPlant 34 -90
turnMoveAndPlant 23 180

define turnMoveAndPlant distance direction
point in direction direction
repeat distance
    move 10 steps
create clone of Tree1
```

```
when I start as a clone
show
go to x: x position of Cat1 y: y position of Cat1
```

Figure 6.4: Screenshot of Zoe's Scratch project that gets 3 points for abstraction and decomposition in Dr. Scratch, showing the cat sprite (left) and the tree sprite (right)

6.2.3 Procedure

The study took place in a small IT suite with each Scratch project loaded on a different PC in random order. Participants were put into groups of 3 by their class teacher based on the teacher's judgement of their Scratch ability. With Group 1 being the most competent and Group 7 being the least competent. Participants were asked to pretend to be teachers, using numbered cards to rank the Scratch projects from best to worst. Each group was introduced to the project specification and behaviour. They were then given a short explanation of custom blocks and clones. The sessions were between 25 and 35 minutes and audio was recorded.

Participants were given four cards, numbered 1 (best), 2, 3 and 4 (worst), to rank the projects using the criteria in Table 6.2. The criteria were introduced in turn so that participants could not infer answers from later questions. They were encouraged to examine each project and come up with a decision as a group. The first criterion was for which project was 'coded the best'. This was deliberately ambiguous to see what factors participants would use to judge 'good' code in Scratch. They were then asked to rate the projects by which was easiest to understand. Then by which had the smallest scripts and least blocks. Finally, participants were asked to alter each project so that the cat navigates and plants trees on a smaller park and then rank them on how easy this was to achieve.

The study took place on the 10th October 2018.

Table 6.2: Scratch project ranking criteria

Criteria	Action	Justification
Best coded	Examine each project	What factors would participants use to judge 'good' Scratch code?
Easiest to understand	Examine each project	Does abstraction make projects easier to understand?
Smallest scripts and least blocks	Examine each project (counting blocks)	Can participants recognise that abstraction has reduced the size of the script and the number of blocks?
Easiest to change	Altering each project in turn	Can participants recognise that abstraction makes a project easier to change?

6.2.4 Ethics and Access to Participants

This study is covered under the same ethics approval as Study 1: a series of studies investigating the effect of visual programming on CT skills (Appendix C). Audio transcripts were anonymised using group ID numbers and original recordings were deleted after they had been transcribed.

6.3 Results

This section will describe the order that each group ranked the projects and their explanations for doing so. The full order that each group placed the projects at each stage can be seen in Table 6.3.

Table 6.3: Scratch project ranking results (the number in brackets is the Dr. Scratch abstraction and decomposition score for the project)

Group Number	Best Coded	Easiest to Understand	Smallest Scripts and Least Blocks	Easiest to Change
1	Zoe (3)	Ava (0)	*Missed*	Zoe (3)
	Alice (2)	Emma (1)	*question*	Alice (2)
	Emma (1)	Alice (2)		Emma (1)
	Ava (0)	Zoe (3)		Ava (0)
2	Zoe (3)	Ava (0)	Zoe (3)	Zoe (3)
	Alice (2)	Emma (1)	Ava (0)	Alice (2)
	Emma (1)	Alice (2)	Alice (2)	Ava (0)
	Ava (0)	Zoe (3)	Emma (1)	Emma (1)
3	Emma (1)	Ava (0)	Zoe (3)	Zoe (3)
	Alice (2)	Emma (1)	Ava (0)	Ava (0)
	Zoe (3)	Alice (2)	Alice (2)	Alice (2)
	Ava (0)	Zoe (3)	Emma (1)	Emma (1)
4	Zoe (3)	Zoe (3)	Zoe (3)	Zoe (3)
	Emma (1)	Emma (1)	Emma (1)	Emma (1)
	Alice (2)	Alice (2)	Alice (2)	Ava (0)
	Ava (0)	Ava (0)	Ava (0)	Alice (2)
5	Alice (2)	Zoe (3)	Zoe (3)	Zoe (3)
	Zoe (3)	Alice (2)	Alice (2)	Alice (2)
	Emma (1)	Emma (1)	Emma (1)	Emma (1)
	Ava (0)	Ava (0)	Ava (0)	Ava (0)
6	Alice (2)	Ava (0)	Ava (0)	Zoe (3)
	Emma (1)	Alice (2)	Emma (1)	Emma (1)
	Ava (0)	Emma (1)	Alice (2)	Ava (0)
	Zoe (3)	Zoe (3)	Zoe (3)	Alice (2)
7	Emma (1)	Ava (0)	Alice (2)	Zoe (3)
	Alice (2)	Emma (1)	Emma (1)	Alice (2)
	Zoe (3)	Alice (2)	Ava (0)	Emma (1)
	Ava (0)	Zoe (3)	Zoe (3)	Ava (0)

6.3.1 Previous Scratch Experience

All groups were asked at the start of the study if they had used Scratch before and then if they had used custom blocks and cloning. All participants had used Scratch, one participant from Group 1 had used custom blocks but had “not managed to get them working”. The participants from the first two groups had used cloning and could explain how it worked.

6.3.2 Best Coded

There was a range of answers for which project was the ‘best coded’. Three groups ranked Zoe’s project (3) as number 1, because “it’s got more advanced coding” and “it’s more complex and modern, it’s only got one tree and it clones it, so it’s also saving space.” Two groups ranked Alice’s project (2) as the best. One group because “it has got more details” when talking about custom block arguments (degrees and distance inputs) and the other because it had four trees that appeared at different times, in comparison with the single tree in Zoe’s project. The final two groups ranked Emma (1) as the best-coded project because it was not “copying trees” and “had the most blocks in the trees”.

It was widely agreed that Ava’s project (0) was the worst coded (6/7 groups). The reasoning for this was that the project was noticeably simpler than the others and did not meet the specification, for example, “there’s nothing in the trees” and “it doesn’t actually plant trees”. One group placed Zoe’s project as the worst because it was the only one with a single tree, “It’s only one tree and I think there should be four trees.”

6.3.3 Easiest to Understand

Most groups (5/7) ranked Ava’s project as the easiest to understand. The main reason for this is that it was “very simple” in comparison with the others and because “there’s no coding in the trees... the trees are already there”. Group 2 suggested that it was the easiest to understand because “all you have to do is paste them down”, “you’ve got everything so you don’t need to add any blocks, you can just copy and paste and add.” The same five groups also ranked Zoe’s project as the most difficult to understand, often because of a lack of understanding of custom blocks and cloning, ”I don’t understand those sorts of codes”. Despite the short introduction to clones at the start of the session, Group 6 asked: “how does one tree end up with four trees?” After a further explanation, one participant stated, “that’s even more complicated.”

The other two groups (4 and 5) ranked Zoe’s project as the easiest to understand. The justification for this was that it “didn’t have the different trees” and that “you can use a different one of these” when referring to the ‘moveTurnAndPlant’ custom block.

6.3.4 Smallest Scripts and Least Blocks

When asked to rank the projects by the smallest scripts and least blocks, each group counted the blocks in each project. Based on this, the correct order should be Ava or Zoe (14 blocks each), Alice (26 blocks) then Emma (30 blocks). Interestingly, participants were unable to see that this was the case, even after counting the blocks in each project. The rankings given for this criterion differed depending on the Scratch ability of participants in each group. Group 1 missed the question, Groups 2 and 3 based their order on the number of blocks and Groups 4 and 5 counted the blocks but did not adjust their order from the previous question. The final two groups (6 and 7) both put Zoe’s project as the worst due to complexity, “because it’s got so much code in.” Several groups had to be reminded that there were blocks in the tree sprites that counted towards the project size.

6.3.5 Easiest to Change

All the groups ranked Zoe's project as the easiest to change (7/7) after adjusting the size of the park in each project. There were two main reasons for this. One was that you did not need to move or adjust the tree sprites; "wherever the cat stops, that's where the tree goes". When asked whether they needed to change the trees in Zoe's project, Group 2 replied "no, because they're going to clone", going on to say, "that one was the easiest to change, and it's always perfect (because of the clones)." The second reason was that they only had to change one number in the cat sprite; "with that one we had to change one number, but with that one we had to change four" and "that one's easiest because its only one step." There was also some evaluation of project quality "it was done good in the first place, you don't have to change much when you're doing it different." It did take some groups several minutes to figure out the best way of completing the task in the first project. Some participants began by changing the numbers in the 'repeat' blocks instead of the 'move' blocks.

There were a range of answers for the project that was the most difficult to change. Ava's project was ranked the worst by 3/7 groups, Emma's project by 2/7 and Alice's by 2/7.

6.4 Discussion

In summary, the best coded and the smallest scripts and least blocks criteria resulted in inconsistent rankings. The simplest project (Ava) was chosen by most groups as the easiest to understand. All groups chose the project with custom blocks and cloning as the easiest to change (Zoe), but only after altering each project in turn. This section discusses these results, giving possible reasons for each finding and then summarising the implications.

The 'best coded' criterion aimed to see what primary school children perceived as 'good code' in Scratch without any indication as to what this meant. As expected, there was a range of different answers. Most groups ranked projects based on their behaviour. This explains why the simplest project (Ava), which produced a slightly different output to the others, was ranked bottom by 6/7 groups.

However, Group 2 chose Zoe's project as the best based on the notion of 'saving space', because it had fewer blocks and fewer sprites. They even went as far as to compare this to the copying and pasting in other projects. This suggests that children in this age group can understand abstraction, even when they have not been formally introduced to custom blocks and cloning in Scratch.

Custom blocks and cloning generally made projects more difficult to understand. The simplest project was widely-chosen as the easiest to understand. Group 2 justified this by saying that copying and pasting made the project easier to understand because it already contained all the blocks needed for the project. This could be one of the reasons that block duplication is so common in Scratch projects. Having primary school children produce programs with duplication, before refactoring it out using procedures, could act as a stepping stone when introducing abstraction and code reuse.

The number of blocks had little effect on what the participants thought of the program. This supports the idea that children must be taught explicitly to recognise duplication and long script smells and be taught how to refactor code to remove them. It highlights the findings discussed in Chapter 5: that children using Scratch frequently duplicate code and produce long scripts, with little understanding of why these smells can result in programs that are difficult to understand, debug and maintain (Hermans & Aivaloglou, 2016).

Finally, having primary school children alter projects was a good strategy for teaching the benefits of procedural abstraction and code reuse, even without them having a full understanding of custom blocks and cloning.

6.4.1 Implications

The variability in the understanding of what makes a well-coded project implies that this skill does not come naturally. Primary school children prefer a copy and paste approach that they find easier to understand. This supports the Scratch project analyses discussed in Chapter 5: children frequently duplicate code even if they have used abstraction in their projects (Robles et al., 2017). This implies that they cannot be expected to appropriately use abstraction if they do not understand the benefits. Yet, the success of the altering project strategy suggests that children can understand the value of abstraction if they are taught how to use it through practical problem-solving tasks and project comparison. They should first be introduced to the correct usage of custom blocks and cloning. Then given tasks that involve producing programs that would benefit from abstraction, with code duplication or long scripts, before being asked to remove these code smells using custom blocks and/or cloning. These tasks would need strict rules to enforce the correct use of abstraction, which would suit a game-based approach using a restricted Scratch environment because rules can be integrated and changed on a level-by-level basis.

Design Concept - Correct Usage
Primary school children should first be introduced to the correct use of custom blocks and cloning before being asked to produce programs containing it.

6.4.2 Limitations

The study was designed to establish the potential for teaching abstraction to children of this age group. It aimed to ensure that the learning goals of the programming game (Chapter 7) were not too difficult for the target age group before development began. This meant that it was somewhat limited in its methodological approach: taking place in a single day and not using a formal experimental design.

6.5 Conclusions

In conclusion, this formative study showed that children age 10 and 11 with limited Scratch experience can recognise the benefits of abstraction in Scratch, but only when asked to alter a project in a way where code reuse was beneficial. There were some indications that children could understand the concept of 'saving space' by using procedural abstraction, compared to copying and pasting. However, copying and pasting was seen as easier to understand by most participants, which may explain why it is common in Scratch projects. There was no consistency in what participants thought constituted a well-coded project. Nor were they able to see that abstraction reduced the number of blocks in a Scratch project.

The findings of this study support the ideas of Armoni (2016) and the research of Gibson (2012) in suggesting that primary school children can learn abstraction in CS through concrete examples and the physical manipulation of information, combined with structured teaching. In this case, showing that duplication and long scripts in Scratch make projects difficult to debug and alter and that these code smells can be removed using custom blocks and clones. Letting children interact with projects that use different blocks to produce the same behaviour was an effective method of getting them to compare programming approaches and see the benefits of code reuse.

This study feeds into the next chapter (Chapter 7), which describes the design and development of a programming game, Pirate Plunder, designed to teach primary school children abstraction skills in Scratch.

Chapter 7

Pirate Plunder - Design and Development

This chapter describes the design and development of Pirate Plunder, a novel block-based educational programming game designed to teach primary school children (age 9 to 11) to use abstraction in Scratch through custom blocks and cloning (Chapter 5), using the observations from the previous chapter (Chapter 6). Two empirical studies were then conducted using the game. These are described in the following two chapters (Chapters 8 and 9).

The chapter begins by describing the aims of Pirate Plunder and the reasoning behind teaching abstraction using a game-based approach. It then gives an overview of the game, a walkthrough of some important elements, before explaining each aspect of the game design. Finally, the chapter describes the iterative development of the game and some of the key changes made throughout this process. The software development of the game, including the technologies and processes used, is explained in Appendix A.

7.1 Aims

The study in the previous chapter (Chapter 6) showed that children age 10 and 11 can understand the benefits of abstraction in Scratch when altering programs themselves. Section 6.4.1 discussed the implications of these findings, suggesting that when teaching abstraction, strict rules should be enforced to make sure that children are using custom blocks and cloning correctly. The idea of introducing these concepts through practical problem-solving lends itself to a game-based approach. Games can be an engaging method of learning new skills, particularly those that involve technology (Boyle et al., 2016). Programming games often involve the player solving puzzles by navigating an object through a grid using block-based or text-based instructions.

Chapter 2 evaluates existing programming games and the different learning approaches used in educational programming tools. Code.org, Lightbot, Gidget and

Dragon Architect all introduce programming concepts and have the player use these concepts to complete a set of levels. In theory, games can avoid the bad programming habits that learners can form in constructionist tools like Scratch by using 'instructional guidance' (Mayer, 2004) to teach children programming concepts (e.g. conditionals, iteration and variables) in a linear level progression. Games can enforce rules that require the player to use concepts correctly and can then change these rules over time.

This was the starting point for Pirate Plunder, to create a programming game where children are introduced to basic Scratch concepts (events and movement) that allow them to complete early levels, before presenting them with problems that require loops (repeat blocks), custom blocks and cloning in a structured difficulty progression. Such a game should be able to successfully demonstrate how to use abstraction in Scratch to reduce code duplication.

The aims of Pirate Plunder feed directly into the design and development of the game:

1. Introduce abstraction through custom blocks and cloning in a way that rationalises and explains its use.
2. Be similar enough to Scratch that skills transfer between the two.
3. Can be played with minimal teacher instruction and interaction.
4. Keep players motivated throughout the delivery of the learning content.
5. Collect data that can be used to evaluate and improve the game.

7.2 Overview

Pirate Plunder is a novel educational block-based programming game that introduces abstraction in a game-based Scratch-like setting. The aim is to teach players to reuse code by having them produce programs with duplicated code, before introducing loops, custom blocks and cloning that enable them to create 'better' solutions with less duplication. The next section gives a walkthrough of what the player would see when they first log in to the game, including a tutorial and a challenge to explain how the game works. Full explanations of the level progression are then given in Sections 7.4.1 and 7.4.2.

The player uses Scratch blocks to program their pirate ship to navigate around a grid, collect items and interact with obstacles (Figure 7.1). They can execute, stop or speed up the execution of their programs using the buttons above the program workspace (7.1A). Levels are divided into 'tutorials' and 'challenges'. Tutorial levels introduce blocks or functionality, with a red parrot character demonstrating how and when to use each block (7.1B). Players then use those blocks to complete a set of

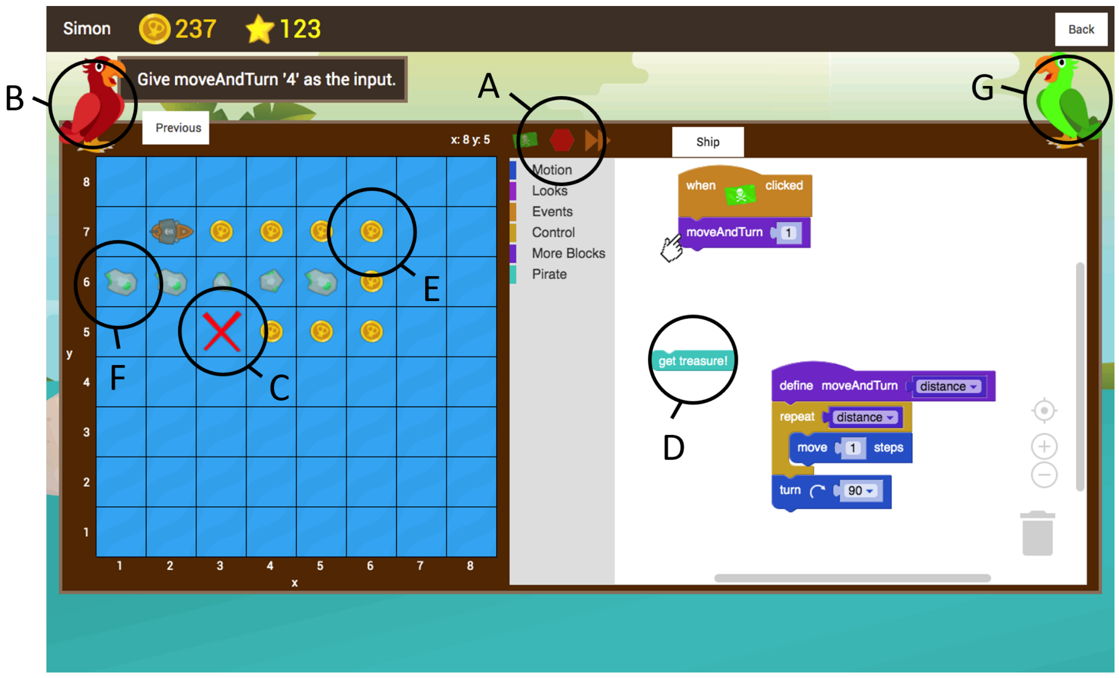


Figure 7.1: Screenshot of Pirate Plunder gameplay, A) Buttons, B) Tutorial parrot, C) X marks the spot, D) ‘get treasure!’ block, E) Map coins, F) Rocks, G) Feedback parrot

challenges before attempting the next tutorial. Levels require the player to navigate to the ‘X marks the spot’ position on the grid (7.1C) and then use the ‘get treasure!’ block (7.1D) to collect a treasure chest that contains coins. Levels also contain ‘map coins’ that can be collected as the player navigates around the grid (7.1E). They must avoid obstacles such as rocks (7.1F) and enemy ships that will sink their ship. The player is assisted by the feedback parrot (7.1G). A star rating is given depending on how many of the map coins the player collected. Collected coins can then be used to purchase items and customise the player’s avatar, which they can then compare with other players.

7.3 Walkthrough

This section describes what happens when the player first logs into the game. Along with an example tutorial (for the ‘repeat’ block) and a corresponding challenge.

7.3.1 Avatar Select

Once the player has logged into their account for the first time. They have to select a starting avatar from two options (Figure 7.2). They can then customise their avatar in the game shop.

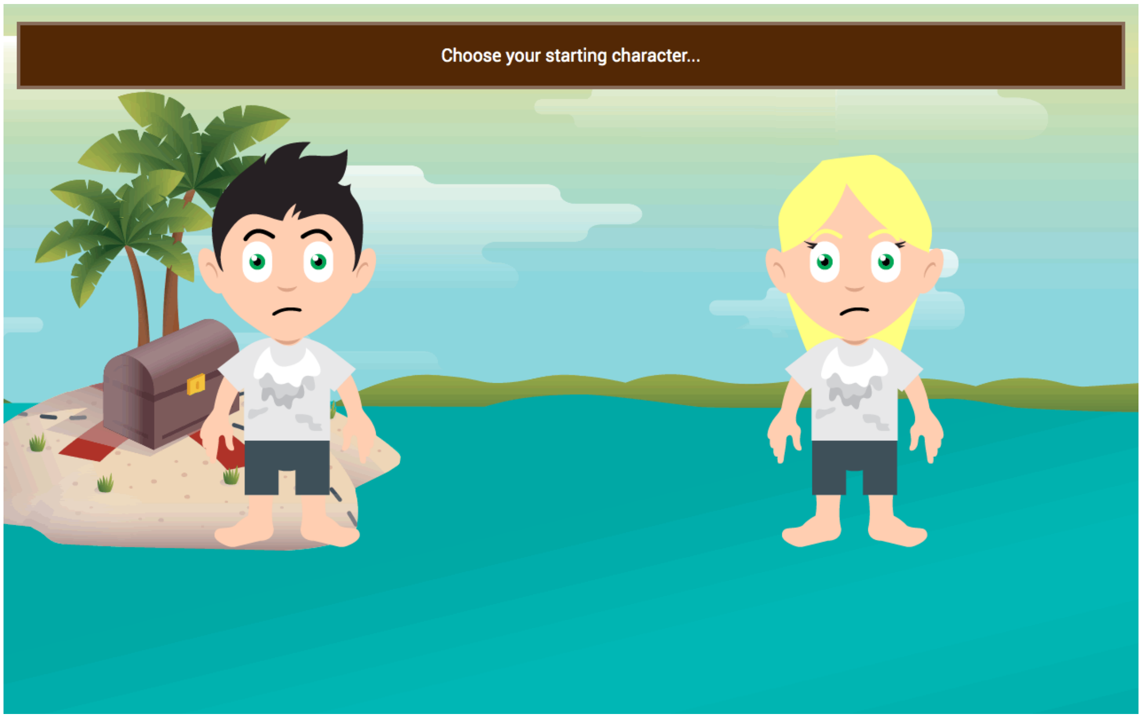
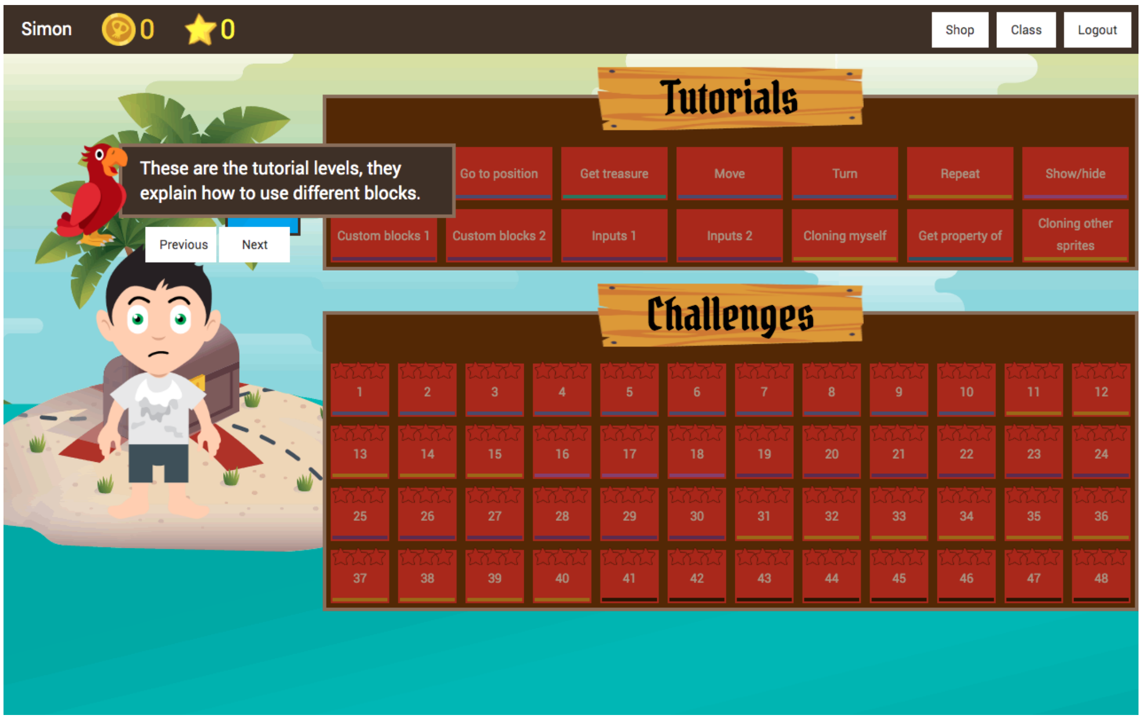


Figure 7.2: Screenshot of the Pirate Plunder character select screen

7.3.2 Level Select

The player is then given a tutorial of the level select screen. This explains how to earn coins and stars, the difference between tutorials and challenges and how to access the shop and class screens.

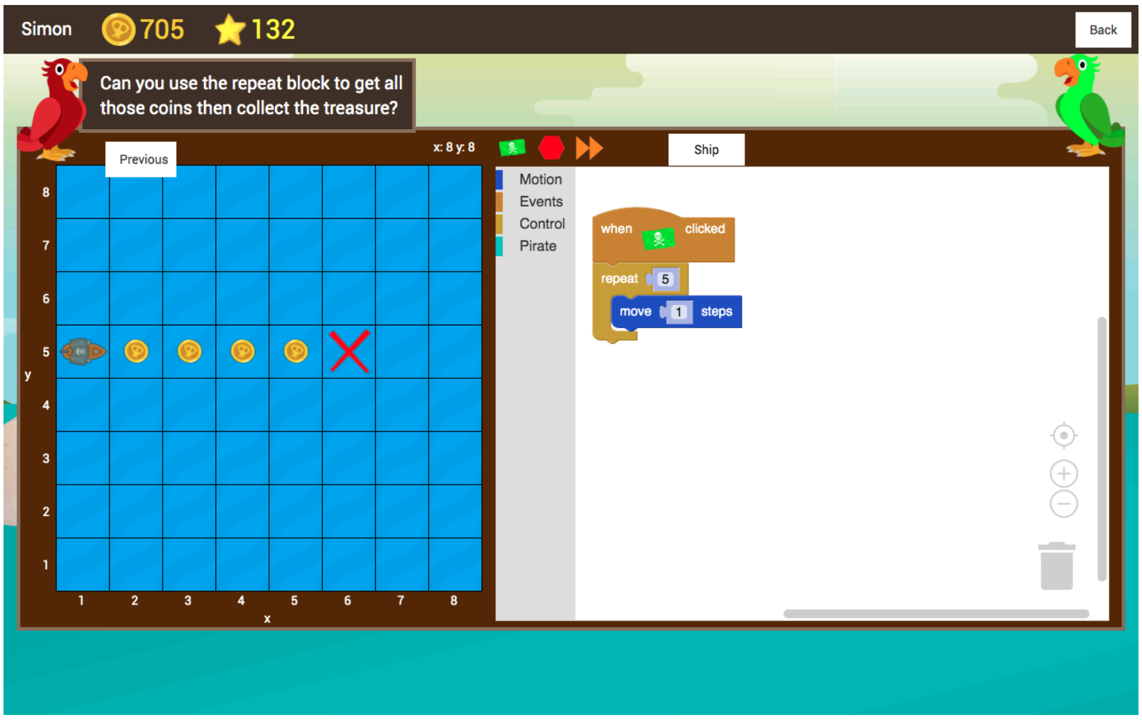


These are the tutorial levels, they explain how to use different blocks.

Figure 7.3: Screenshot of the Pirate Plunder level select tutorial

7.3.3 Tutorials

Tutorials introduce blocks or functionality to the player. Each tutorial begins with the red parrot telling the player what they will be learning. They are then directed through a series of actions that they must complete before they can execute the program and complete the level. Figure 7.4 shows the program in the ‘repeat’ tutorial that the player will have produced by the final instruction. The program will execute when the green flag above the workspace is clicked. It will move the ship five spaces, one step at a time, which will collect all the map coins and get the ship to the treasure. The player needs to complete the level by adding the ‘get treasure!’ block to the end of their program, to collect the treasure chest when they are over the ‘X marks the spot’.



Can you use the repeat block to get all those coins then collect the treasure?

```
when [flag] clicked
repeat 5
    move 1 steps
```

Figure 7.4: Screenshot of the Pirate Plunder ‘repeat’ block tutorial

7.3.4 Challenges

Challenges develop the ability of the player to use the blocks or functionality that they have been taught in the corresponding tutorial. There are several challenges for each tutorial that get progressively more difficult. Once the player has completed all of them, they are able to attempt the next tutorial. Figure 7.5 shows a ‘repeat’ challenge with the correct solution. The program moves the ship to collect all the available map coins before reaching ‘X marks the spot’, using two repeat blocks and a turn.

Challenges differ from tutorials in that they have a block limit and a reset button and do not have the red parrot, yet the underlying aim of each level is the same. The player cannot execute their program if they go over the block limit. The reset button can be used to empty the program or reset it to its original state.

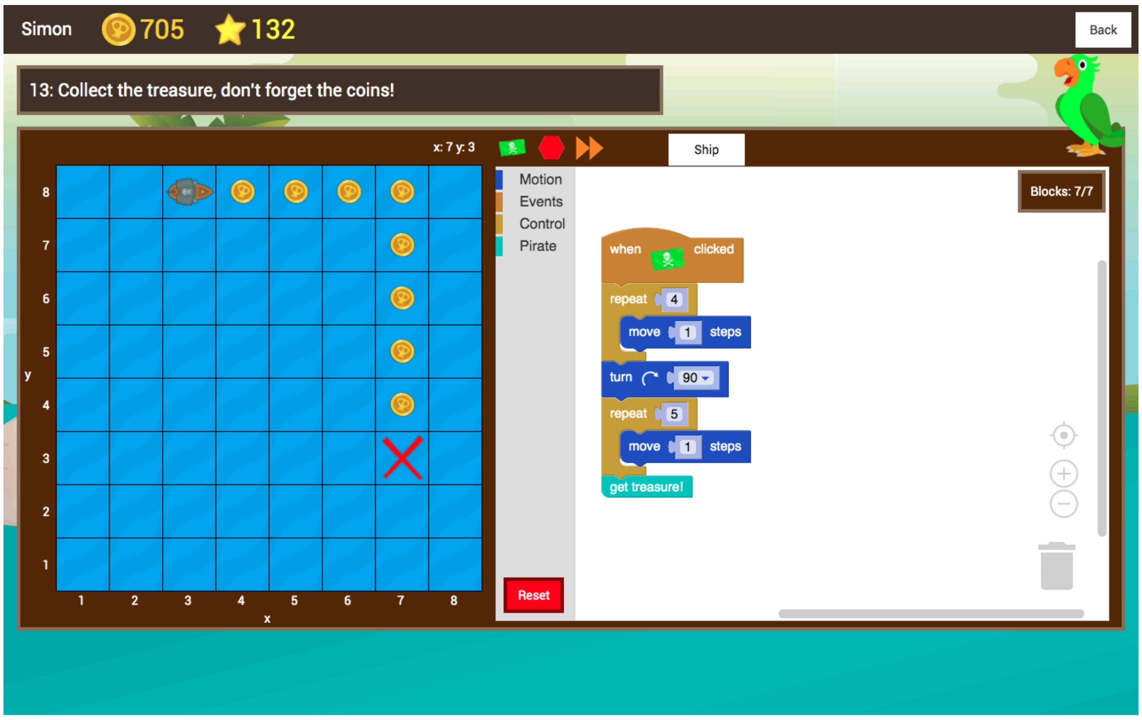


Figure 7.5: Screenshot of a Pirate Plunder 'repeat' block challenge

7.4 Game Design

This section begins with a level-by-level explanation of the game, it then explains the rationale behind different aspects of the game design: difficulty progression, Scratch integration, reward system, tutorials, in-game feedback and hints, customisation, administration section, analytics and sounds. These are described as Pirate Plunder was at the start of Study 3 (Chapter 8). The subsequent section (Section 7.5) explains how the game was developed up to that point using an iterative development process.

7.4.1 Level-by-Level

This section describes several example levels in Pirate Plunder, highlighted in Figure 7.6. This explains in detail how the game works and how the learning content is introduced. It starts off with an early tutorial level (7.6A) and a 'turn and move' challenge (7.6B). It then explains the final 'repeat' challenge (7.6C), the final 'show/ hide' challenge (7.6D), custom block tutorials (7.6E), inputs levels (7.6F and 7.6G) and cloning levels (7.6H and 7.6I). The colours underneath the levels correspond to the colour of their block category, giving the player an idea of how many challenges are required before they can attempt the next tutorial.

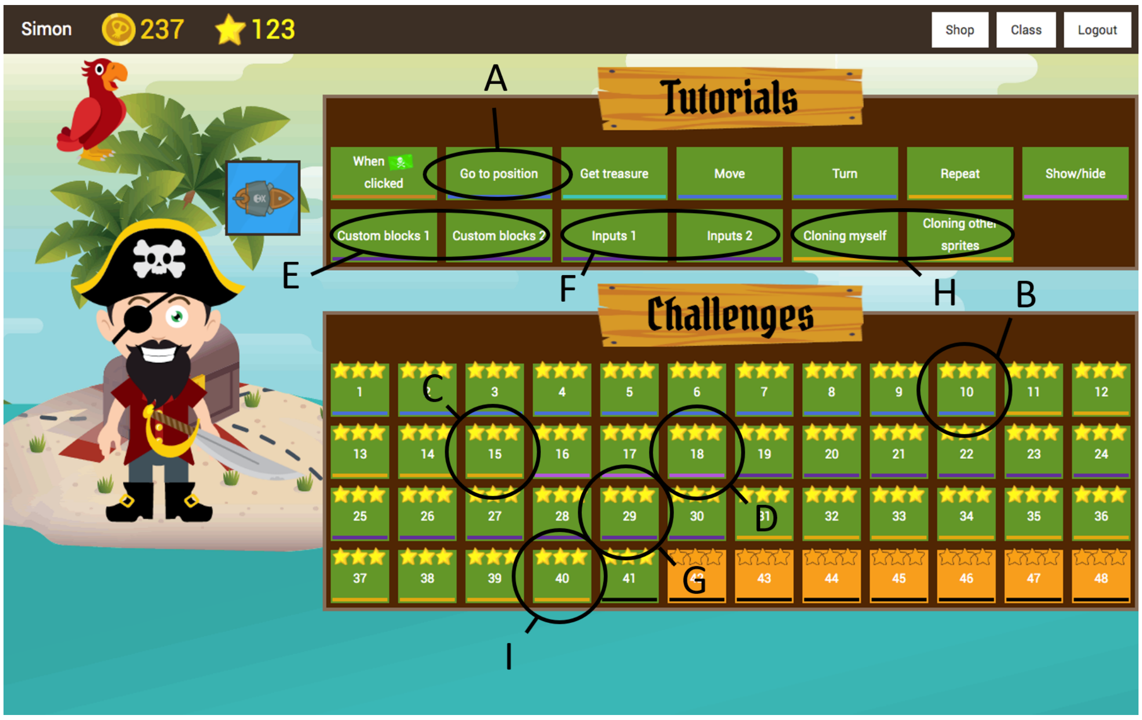


Figure 7.6: Screenshot of Pirate Plunder level select, the highlighted levels are described in the level-by-level section

Go To Position Tutorial

The ‘go to position’ tutorial is the second level in the game. It is part of three initial tutorials, along with ‘when green flag clicked’ and ‘get treasure’, that introduce the player to the fundamental blocks used in the game. The first challenge is not unlocked until these are complete. The tutorial introduces grid positions and the ‘go to position’ block. The block is only used in the first two challenges and is then not used again until the cloning levels (where it is only available in the cannonball). This is because it makes it too easy for the player to navigate to the treasure.

The first stage of the tutorial gets the player to add the ‘go to position’ block to the workspace (Figure 7.7). They are asked by the tutorial parrot (7.7A) to open the ‘Motion’ block category and then to drag the block into the program so that it connects with the ‘when green flag clicked’ block. The pointer (7.7B) directs the player and demonstrates certain functionality, such as ‘click’ or ‘click and drag’. In tutorials, the instructions cannot advanced until the player has completed the required action. The player then cannot run the program until they have finished the instructions (7.7C).

The second stage of the tutorial explains the grid coordinates (Figure 7.8), including the x and y-axis and the bottom-left (1, 1) and top-right (8, 8) coordinates (7.8A and 7.8B). It then shows how the player can find out coordinates of each grid position using the indicator above the grid (7.8C).

The player is then asked to demonstrate an understanding of the coordinate system by selecting the correct grid position for three different sets of coordinates (Figure 7.9A). The green feedback parrot will tell them if they get this wrong (7.9B).

Once the player has finished the coordinate stage, they must then use the block to move the ship to the coin (Figure 7.10). This involves finding the position of the coin (7.10A) and putting these numbers into the ‘go to position’ block inputs (7.10B). They then press the green flag (7.10C) and the ship will move. The level is complete once the player has collected the coin.

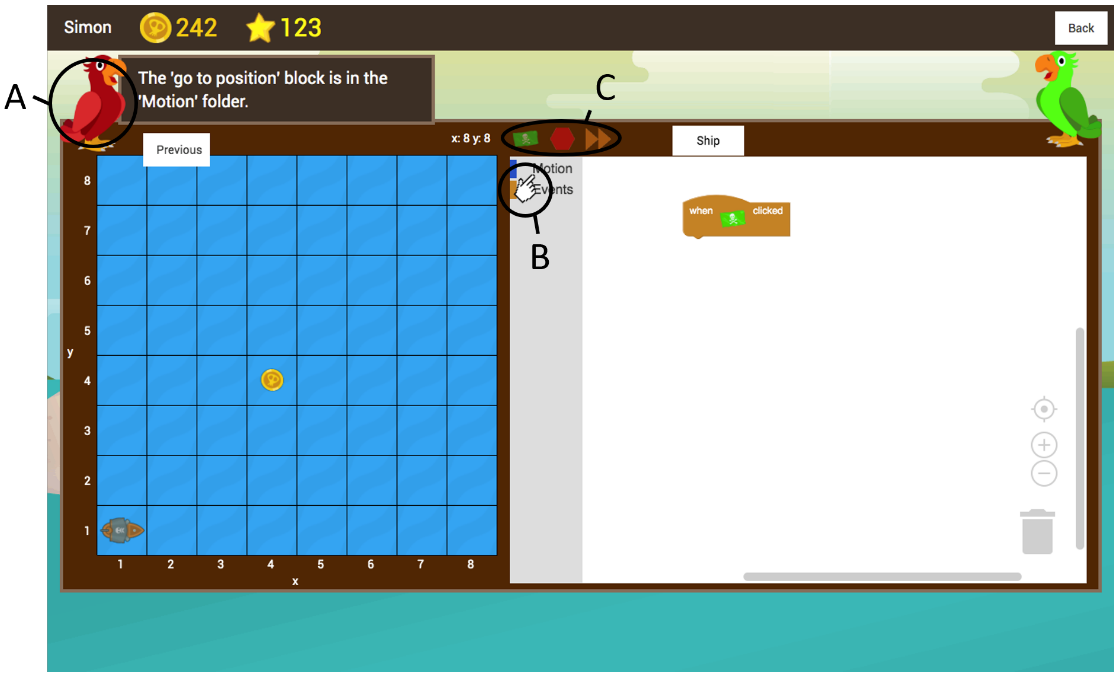


Figure 7.7: Screenshot of the first instruction on the Pirate Plunder ‘go to position’ tutorial, A) Tutorial parrot, B) Pointer, C) Greyed out buttons

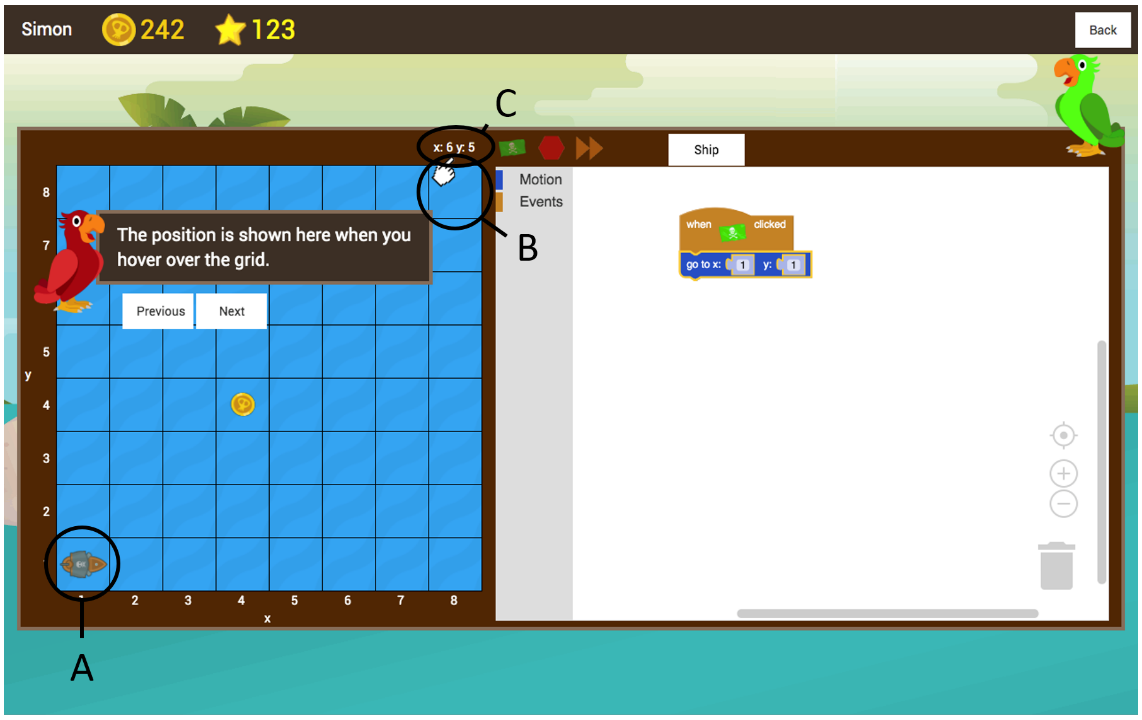


Figure 7.8: Screenshot of the second stage of the Pirate Plunder ‘go to position’ tutorial, A) Bottom-left: 1, 1, B) Top-right: 8, 8, C) Grid position indicator

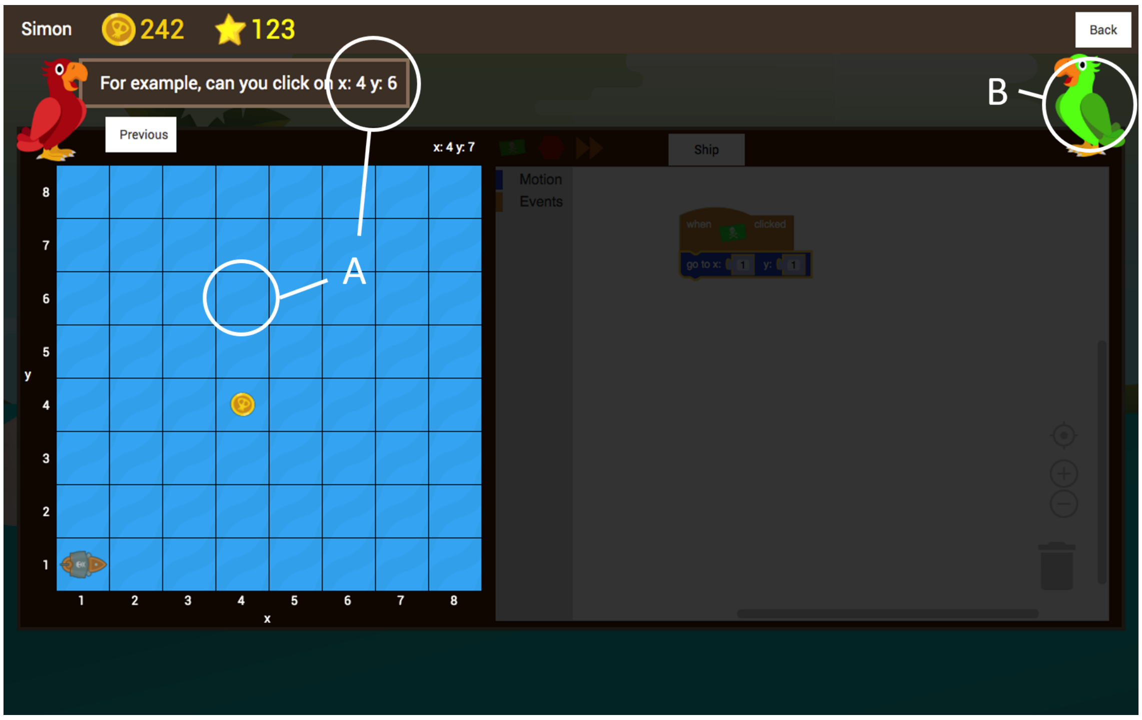


Figure 7.9: Screenshot of the third stage of the Pirate Plunder 'go to position' tutorial, A) Example coordinate demonstration, B) Feedback parrot

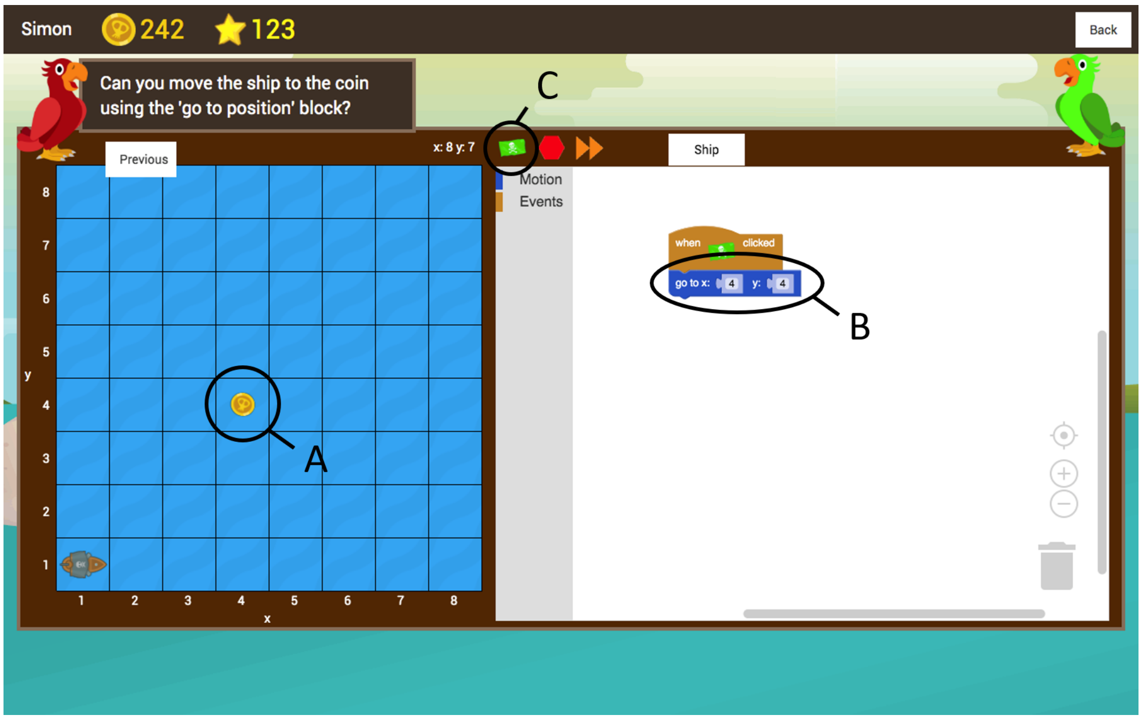


Figure 7.10: Screenshot of the fourth stage of the Pirate Plunder 'go to position' tutorial, A) Position of the coin, B) Entering coordinates into the 'go to position' block inputs, C) Green flag to execute the program

Move and Turn Example Challenge

This example is the final 'turn' challenge before the 'repeat' block is introduced (Figure 7.11). This challenge uses most of the blocks that the player has been introduced to so far ('when green flag clicked', 'get treasure', 'move' and 'turn'). Each challenge has the level number and a short description of the aim of the level in the top left of the screen (7.11A). The green parrot in the top right gives the player feedback throughout the level (Section 7.4.6) (7.11B). Challenges have a block limit (7.11C) that the solution cannot exceed. The player can clear their solution (or bring back the starting solution) using the 'reset' button (7.11D).

The ideal solution to this level is to turn immediately to avoid the rock, then to stop on each coin until the ship reaches the 'X marks the spot', turning left 90 degrees after seven 'move 1 step' blocks. The player must collect all the map coins to get a three star rating on the level (Section 7.4.4). Only blocks connected to the 'when green flag clicked' block will run when the green flag is clicked. Like Scratch, blocks can be in the program workspace that are not executed. The player can zoom in and out of the workspace if necessary (7.11E). The inputs for the 'turn' blocks are 90 by default and are limited to 0, 90, 180 and -90, meaning that the ship stays within the grid lines. The input for the 'move' block can be any integer. Blocks can be duplicated by right-clicking on them and selecting the 'duplicate' option and can be deleted by dragging them out of the program workspace or into the bin (7.11F).

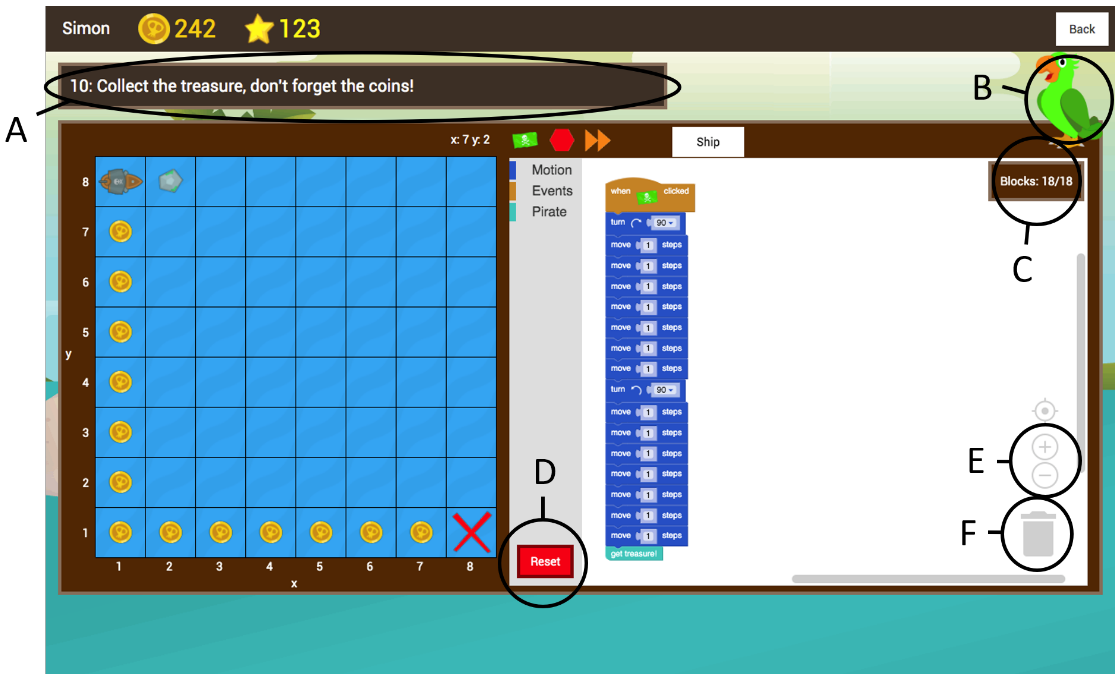


Figure 7.11: Screenshot of the solution to the final Pirate Plunder 'turn' challenge, A) Level number and challenge instructions, B) Feedback parrot, C) Block limit, D) Reset button, E) Program zoom, F) Block bin

Once the player produces a solution that gets the ship to the 'X marks the spot' and uses the 'get treasure!' block, a treasure chest appears on the screen that gives the player a random number of coins between 1 and 15. They are then given a star rating for their performance on the level before being returned to the level select screen.

The rationale for this challenge is to get the player to duplicate a 'move 1 steps' block. This is a time-consuming process and uses the maximum number of blocks for the level (18). The next level is the 'repeat' tutorial that introduces loops, allowing the player to only use a single 'move' block inside a 'repeat' to collect several coins in a line, meaning that this challenge could be completed in eight blocks, instead of 18.

Repeat Blocks

Between the final 'turn' challenge and the example in Figure 7.12, the player has completed the 'repeat' tutorial and four 'repeat' challenges of increasing complexity. This example is the final 'repeat' challenge before custom blocks are introduced (with the show and hide levels in between). 'Repeat' blocks work by repeating the blocks inside them the number of times specified in the input. Their main use in Pirate Plunder is to collect lines of coins using a single 'move 1 steps' block.

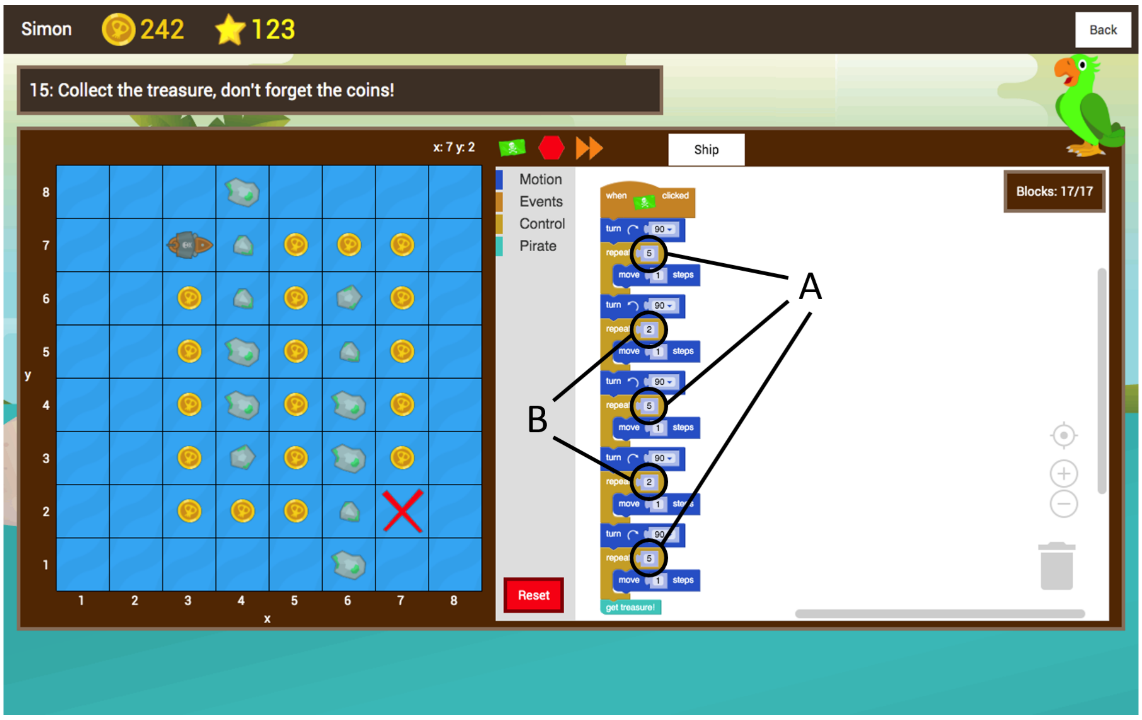
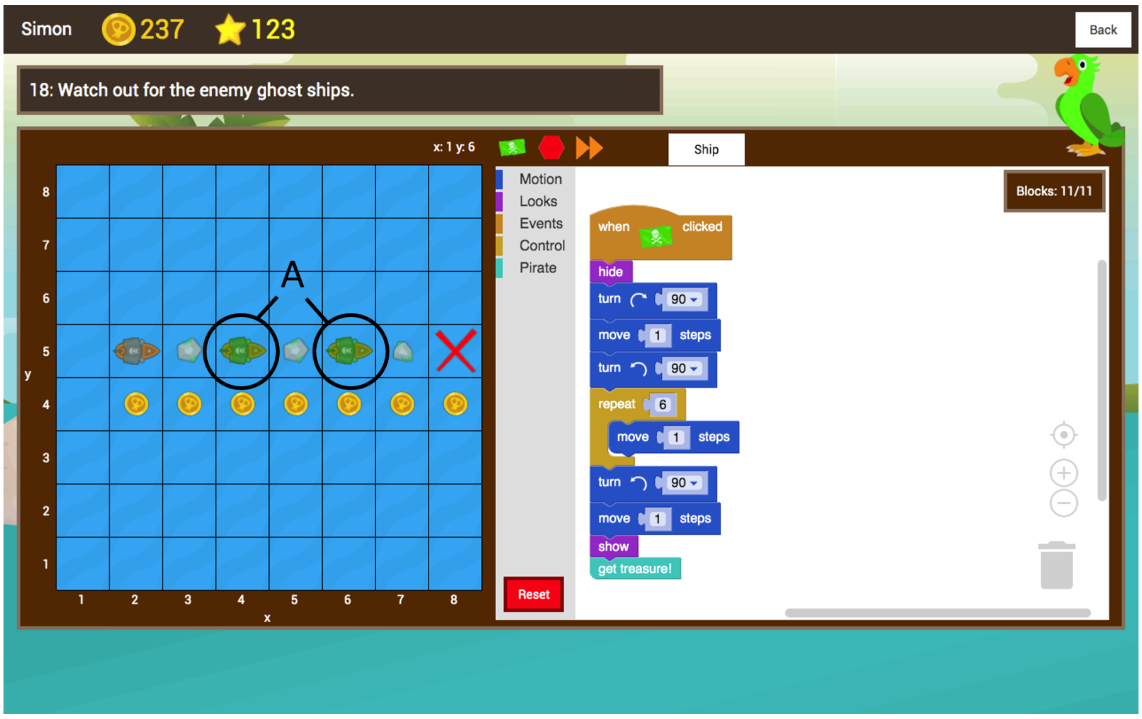

15: Collect the treasure, don't forget the coins!

Figure 7.12: Screenshot of the solution to the final Pirate Plunder 'repeat' challenge, A) Duplicated 'repeat' blocks to collect lines of five coins, B) Duplicated 'repeat' blocks to collect lines of two coins

This challenge has five lines of coins, three that are 5 coins in length and two that are 2 coins in length. The optimal solution (to achieve three stars on the level) has duplicated 'repeat' blocks to collect all of the coins in sequence (7.12A and 7.12B), with 'turn' blocks of different directions in between. This duplication can be removed using custom blocks.

Show and Hide Blocks

'Show' and 'hide' blocks are introduced after 'repeat' blocks. They are in Pirate Plunder because they can be used in Scratch to hide and show cloned sprites (as shown in the best solution in study 2 (Figure 6.4)). In Pirate Plunder, they are used to hide from enemy ‘ghost’ ships (Figure 7.13A) that will shoot at the player’s ship if it goes within range (one grid space horizontally or vertically). These cannot be hit by the player’s cannonballs (when they are introduced in the cloning levels). When hidden, players can still collect map coins, but cannot collect the treasure chest. This means that both ‘show’ and ‘hide’ blocks must be used to complete the level.



```
when [flag] clicked
hide
turn ↻ 90
move 1 steps
turn ↺ 90
repeat 6
    move 1 steps
turn ↺ 90
move 1 steps
show
get treasure!
```

Figure 7.13: Screenshot of the last Pirate Plunder ‘show and hide’ challenge, A) Enemy ghost ships

Custom Blocks

The custom block and inputs tutorials are both split into two. The first tutorial in each converts the solution to a challenge that the player has already completed to use either custom blocks or inputs (Figure 7.14 shows this for custom blocks, converting the ‘repeat’ challenge solution from Figure 7.12 using the extract method described in Section 5.3.1). The second tutorial then walks the player through creating a custom block (with or without inputs), before leaving them to complete the rest of the level on their own.

As with the other tutorials, instructions only move on once the player has completed each action correctly. Custom blocks tutorial 2 has the player open up the ‘More Blocks’ category and press ‘Make a Block’, which opens up the block creation modal window (Figure 7.15A). They then have to name the block correctly (‘move3AndTurn’) and are unable to close the modal window until they have (7.15B). The tutorial has them define the ‘move3AndTurn’ block to move three steps sequentially and turn right (Figure 7.16) (7.16A). This involves directing them to each block category, having them drag blocks to the right place and setting the correct input values. The player is then directed to add a 'when green flag clicked' block and to use their custom block underneath (differentiating between the defining block and the block itself). Three 'move3AndTurn' blocks are needed because the level requires the operation to be performed three times (7.16B).

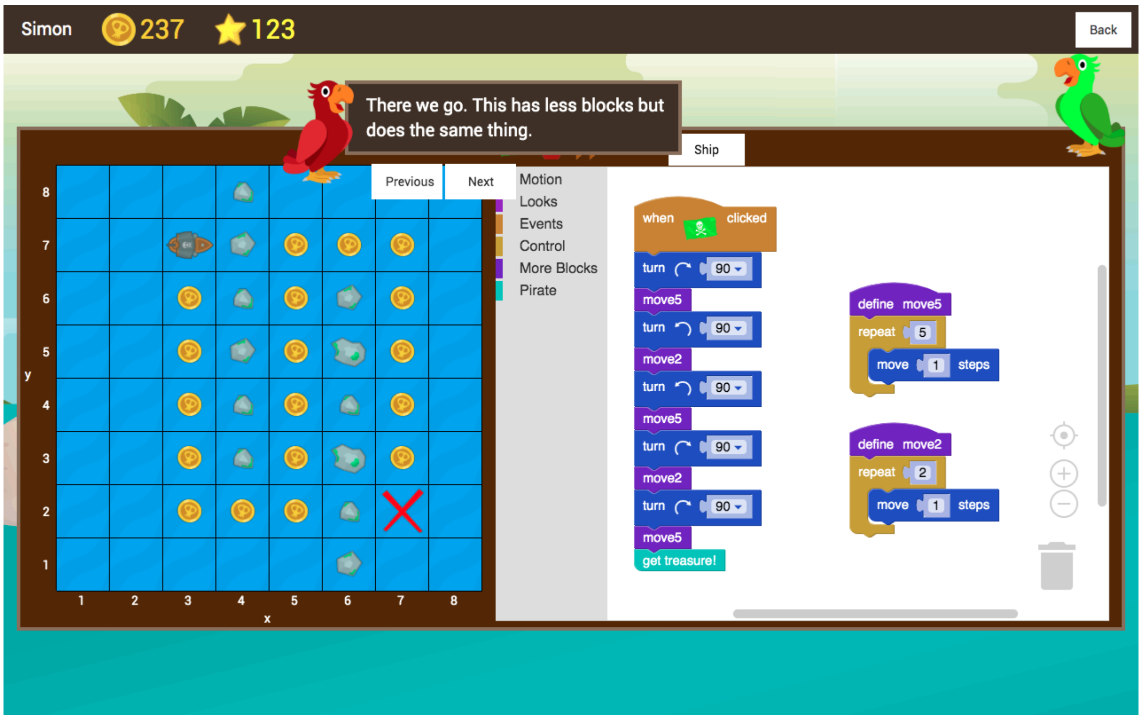


Figure 7.14: Screenshot of the Pirate Plunder 'custom blocks' tutorial 1 that converts the solution in Figure 7.12 to use custom blocks for the duplicated 'repeat' blocks

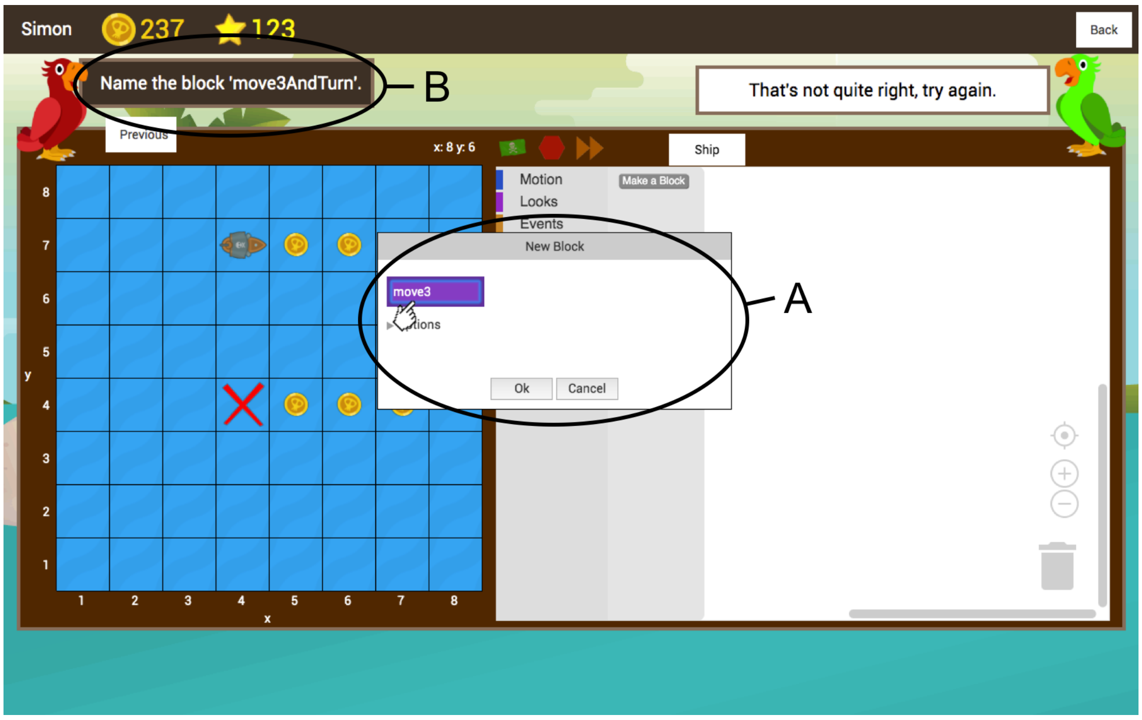


Figure 7.15: Screenshot of the Pirate Plunder 'custom blocks' tutorial 2, A) Custom block creation window, B) The tutorial parrot specifying the custom block name

The player then has to use custom blocks for the next set of challenges. Custom blocks are required (and validated) so that the player cannot keep completing levels using the duplicated 'repeat' strategy from Figure 7.12. Most of these challenges require two custom blocks that are duplicated with only the 'repeat' block input changing for the distance (e.g. move2 and move5). This duplication can be removed by using a single custom block with inputs for the changing values.

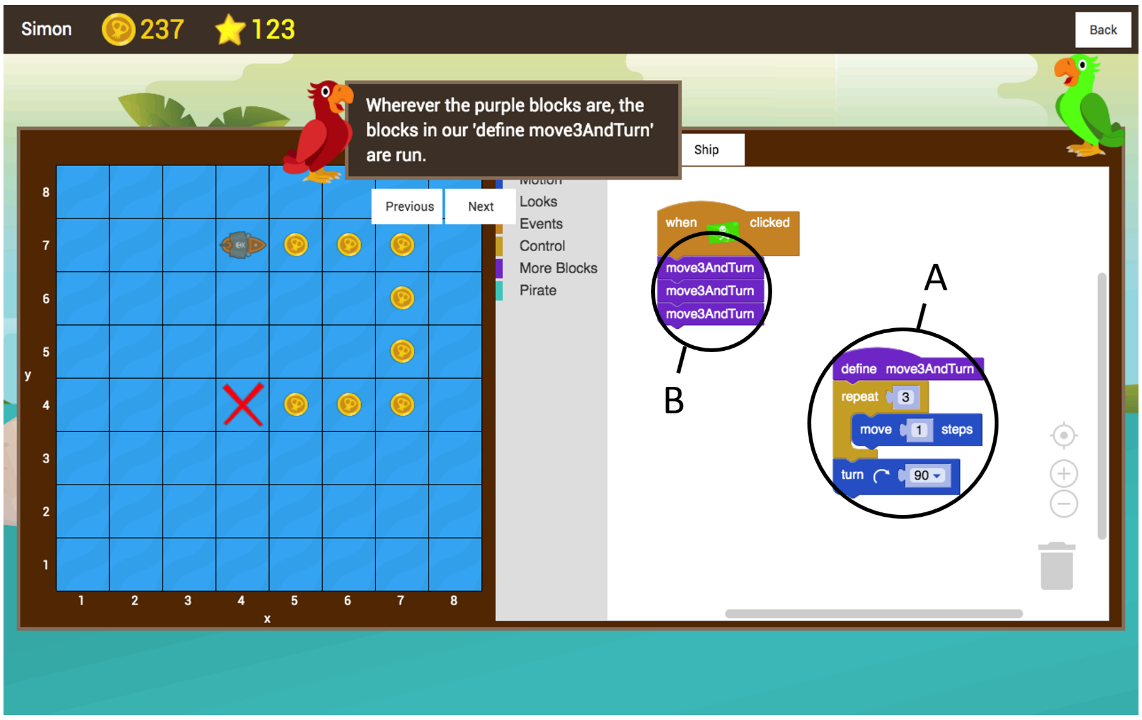


Figure 7.16: Screenshot of the Pirate Plunder 'custom blocks' tutorial 2 showing an almost complete solution, A) Defining the 'move3AndTurn' block, B) Using the 'move3AndTurn' block three times to get to the treasure

Inputs

The inputs tutorial 1 takes the solution from level 23 (this is indicative of all the custom block levels) where the player has used two custom blocks that only differ by their 'repeat' block input ('move3' and 'move5'). It then converts them into a single custom block with an input (parameter) for the 'distance' or number of times the 'move 1 steps' block is repeated (Figure 7.17) (7.17A and 7.17B). The 'moveAndTurn' block then takes the 'distance' as its input (7.17C).

Inputs 2 works similarly to custom blocks 2. It has the player create a custom block through the 'More Blocks' 'Make a Block' button called 'moveAndTurn'. It asks the player to add a 'number input' using the custom block options (Figure 7.18) (7.18A and 7.18B). After this, they are asked to add a 'repeat' block and duplicate the input (using the right-click menu) and drag the duplicated input into the 'repeat' block input. After adding a 'move 1 step' block to the 'repeat and a turn right after the 'repeat', they are then asked to add a 'when green flag clicked' block and one of their 'move-

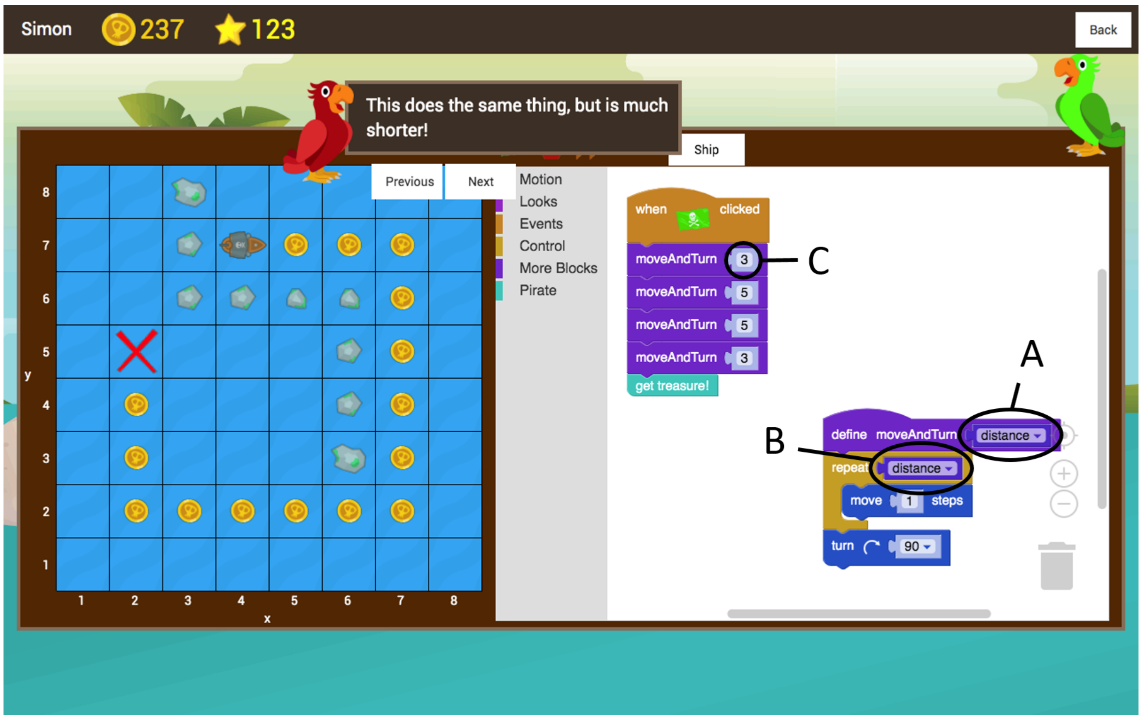


Figure 7.17: Screenshot of the Pirate Plunder ‘inputs’ tutorial 1 that converts a ‘custom blocks’ solution from a previous challenge to use inputs, A) ‘distance’ input, B) Using the ‘distance’ input for the number of repeat iterations, C) The input ‘distance’ passed to the ‘moveAndTurn’ block

AndTurn’ blocks to it (Figure 7.1). The tutorial then explains how the ‘distance’ input is used in the define block for the number of iterations. The player must use two more ‘moveAndTurn’ blocks, with different input values, to complete the level.

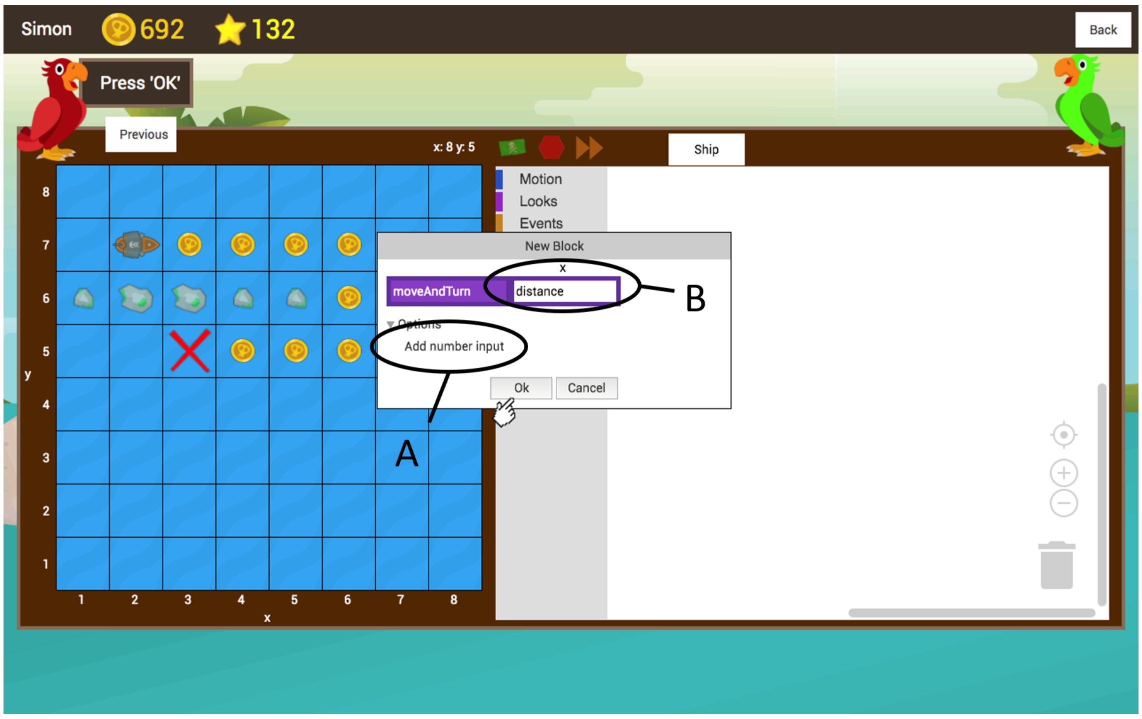
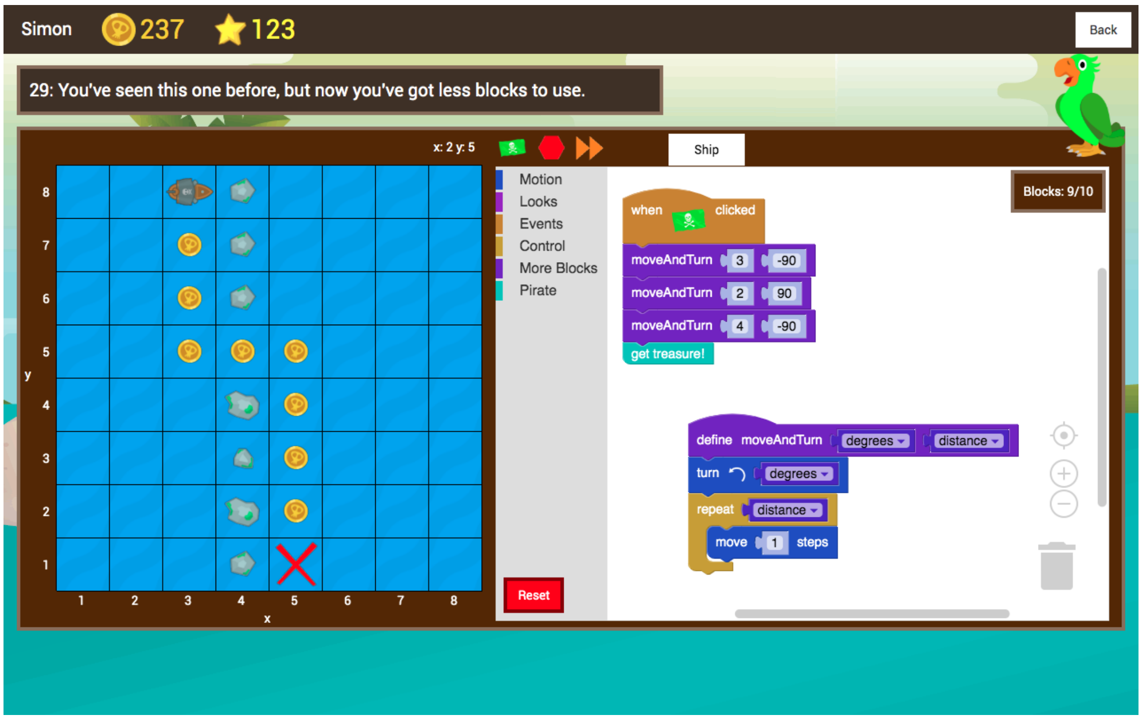


Figure 7.18: Screenshot of the Pirate Plunder ‘inputs’ tutorial 2, adding a number input, A) Add number input option, B) ‘distance’ input

The input challenges progress from using one input ('distance') to two inputs ('distance' and 'degrees'). The degrees input is used to rotate the ship in different directions, which allows the custom block to be used in more cases (Figure 7.19 shows an example of this). This further reduces the total blocks needed to complete the level. Using two inputs is enforced because players are asked to complete the same map again (in another challenge) but with fewer blocks. Pirate Plunder makes it possible to complete some levels several blocks under the maximum block count. For example, in Figure 7.19 the player has moved the 'turn' block to be executed first in 'moveAndTurn' because the ship needs to be turned before it can move. Another solution would be to have a 'turn' block after the 'when green flag clicked' block, then having the last 'moveAndTurn' block turn the ship 0 degrees, which would use an extra block.



```
when [flag] clicked
moveAndTurn 3 -90
moveAndTurn 2 90
moveAndTurn 4 -90
get treasure!

define moveAndTurn degrees distance
turn ↺ degrees
repeat distance
    move 1 steps
```

Figure 7.19: Screenshot of the solution for the final Pirate Plunder 'inputs' challenge

Cloning

Pirate Plunder contains two cloning tutorials: 'cloning myself' and 'cloning other sprites'. 'Cloning myself' has the player create a copy of their ship that they can use to collect another line of coins. 'Cloning other sprites' introduces the cannonball sprite, having the player clone it at the position and in the direction of the ship, then move it to 'fire' it into some floating boxes that contain coins and block off parts of levels. There are three 'cloning' blocks in the game (as in Scratch): 'when I start as a clone', 'create clone of' and 'delete this clone'.

In the cloning myself tutorial (Figure 7.20), the player is first asked to add a 'when green flag clicked' block to the program, then a 'create clone of' block and a 'move 1 step' block (7.20A). They then add 'when I start as a clone' (7.20B), 'go to position' (set to 2, 2) and ‘delete this clone’ blocks to a separate script. When run, this creates a clone of the ship that moves to the start of the other line of coins, then deletes it and moves the original ship sprite 1 step. The player must then figure out how to collect all the coins (by adding ‘repeat’ blocks to both scripts), before collecting the treasure chest.

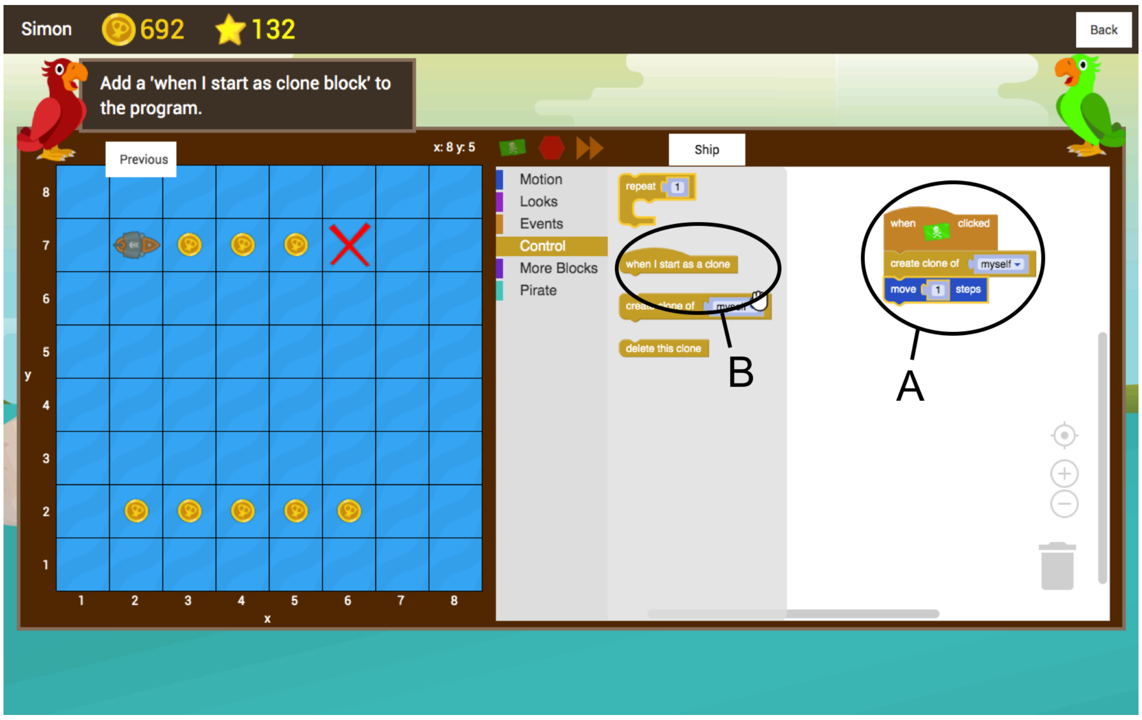
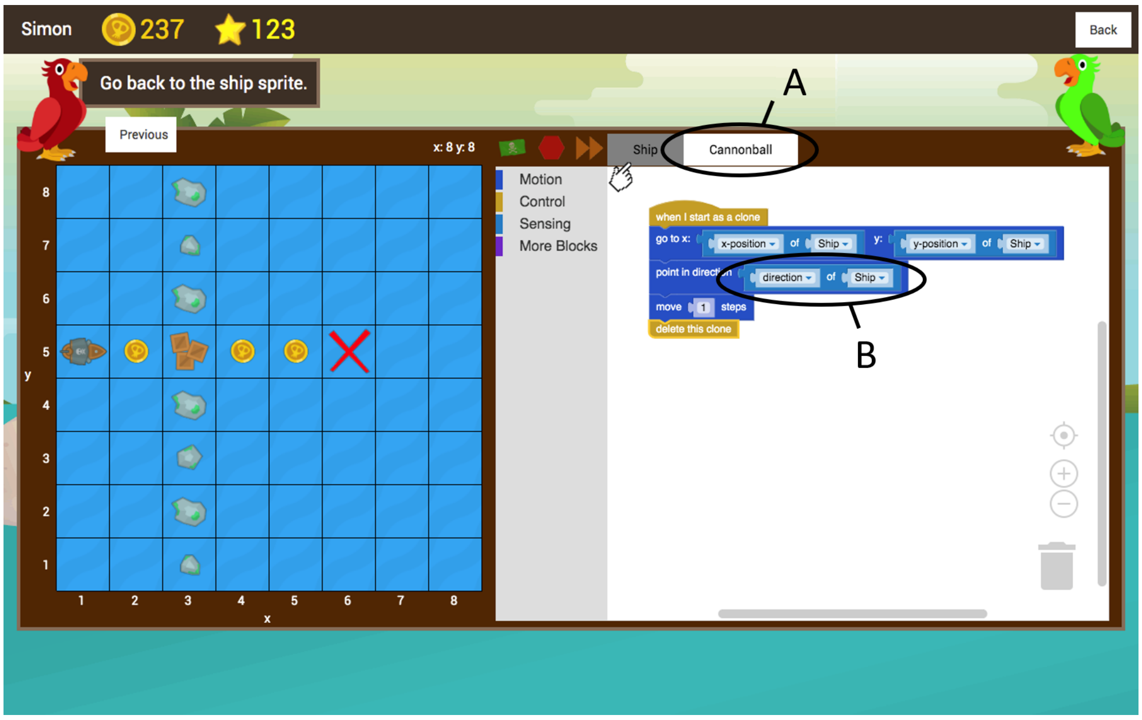


Figure 7.20: Screenshot of the ‘cloning’ blocks in Pirate Plunder, A) The blocks for the original sprite. B) Adding the ‘when I start as a clone’ event to the program indicated by the click and drag pointer

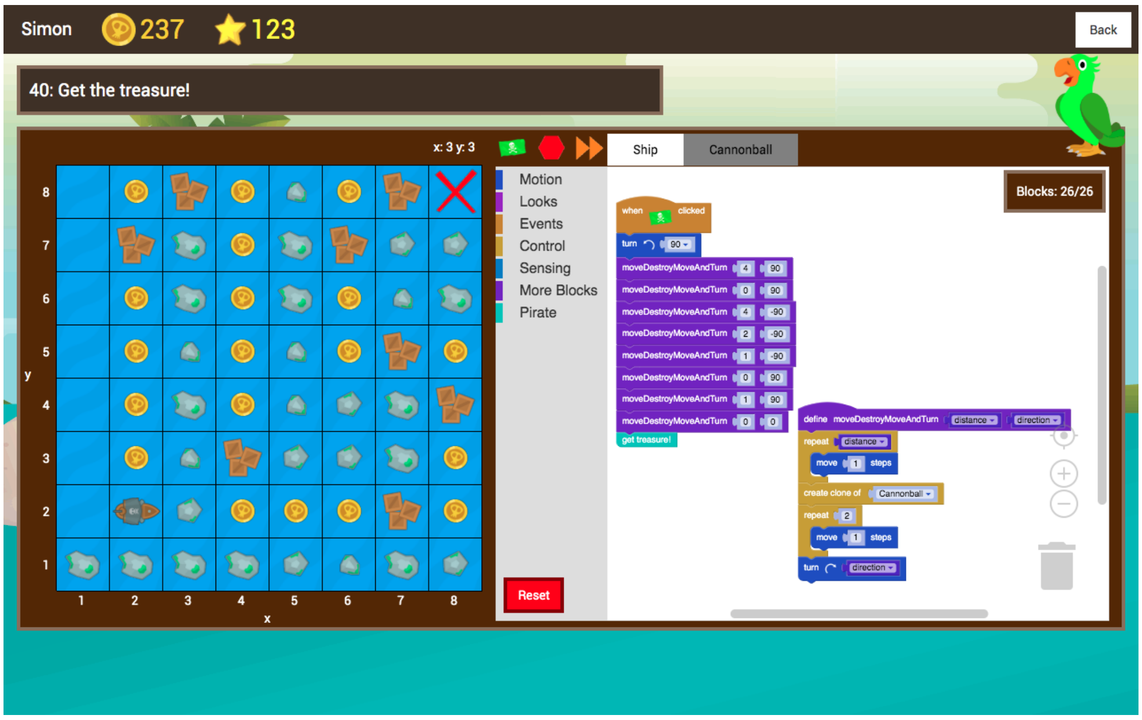
The cloning other sprites tutorial (Figure 7.21) adds a ‘cannonball’ sprite (7.21A) and the ‘property of’ block (in the ‘Sensing’ category). The player adds a ‘when I start as a clone’ block to the cannonball sprite. They then attach a ‘go to position’ block and use the ‘property of’ block to move the cannonball to the position of the ship (it starts off-screen), before pointing it in the same direction as the ship using ‘point in direction’ (7.21B). The player then adds a ‘move 1 step’ block to fire the cannonball from the ship’s current location, before deleting the cloned sprite. This script allows the player to fire cannonballs from different places on the level by creating a clone of the cannonball sprite. The cannonball sprite has a limited set of blocks available.

The cloning challenges progress from using a single cannonball to multiple cannonballs, then to multiple cannonballs with custom blocks (Figure 7.22 shows the final cloning level). General levels then introduce extra complexity, such as having to clone the ship to access blocked off parts of a level. These are not part of the difficulty progression and were added to occupy players who had completed the game during Studies 3 and 4 (Chapters 8 and 9).



```
when I start as a clone
go to x: (x-position of Ship) y: (y-position of Ship)
point in direction (direction of Ship)
move (1) steps
delete this clone
```

Figure 7.21: Screenshot of the cannonball sprite in the Pirate Plunder 'cloning other sprites' tutorial, A) Cannonball sprite selection, B) A 'property of' block used to get the direction of the ship sprite



```
when [flag] clicked
turn ↶ (90)
moveDestroyMoveAndTurn (4) (90)
moveDestroyMoveAndTurn (0) (90)
moveDestroyMoveAndTurn (4) (-90)
moveDestroyMoveAndTurn (2) (-90)
moveDestroyMoveAndTurn (1) (-90)
moveDestroyMoveAndTurn (0) (90)
moveDestroyMoveAndTurn (1) (90)
moveDestroyMoveAndTurn (0) (0)
get treasure!

define moveDestroyMoveAndTurn (distance) (direction)
repeat (distance)
    move (1) steps
create clone of (Cannonball)
repeat (2)
    move (1) steps
turn ↷ (direction)
```

Figure 7.22: Screenshot of the last 'cloning' level in Pirate Plunder

7.4.2 Difficulty Progression

Section 7.4.1 describes how different blocks are introduced in Pirate Plunder. This section explains the difficulty progression and the mechanics used to motivate the player to use the taught functionality. The difficulty progression aims to introduce abstraction using custom blocks and cloning in a way that rationalises its use (Aim 1), allows the game to be played with minimal teacher instruction (Aim 3) and keeps players motivated throughout (Aim 4).

Pirate Plunder contains 40 challenges and 13 tutorials that are split into four difficulty stages: statements (event and motion blocks), loops (repeat blocks), procedures (custom blocks and inputs) and instances (cloning) (Table 7.1). The latter three stages introduce a different technique for abstraction and code reuse, to help the player recognise how they could reduce duplication in previous levels. The number of levels in each section corresponds to the difficulty of that section, with harder concepts having more challenge levels. The difficulty progression was developed over the iterative development process (Section 7.5). It is the same for every player and is not scaffolded based on individual ability or performance.

Table 7.1: Pirate Plunder difficulty progression

Stage	Block (Tutorial)	Challenge Requirements	Number of Challenges
Statements	When green flag clicked Go to position Get treasure!	Move to a grid position and collect the treasure chest.	2
	Move	Move in a single direction and collect the treasure chest.	3
	Turn	Change direction to avoid obstacles.	5
Loops	Repeat	Use loops to reuse blocks.	5
	Show/hide	Hide and show the ship to avoid being attacked by enemy ships.	3
Procedures	Custom blocks	Create and use custom blocks to reuse sets of blocks.	6
	Inputs	Create and use custom blocks with number inputs for further reuse.	6
Instances	Cloning (myself)		
	Cloning (other sprites)	Clone a cannonball sprite to destroy floating debris and access other parts of the map.	10

Pirate Plunder combines 'process constraints': increasing the number of features (or blocks) that the learner can control as they progress through the game, with 'explanations': specifying exactly how to perform an action (on tutorial levels) (Lazonder & Harmsen, 2016). Blocks are introduced after the appropriate tutorial has been completed, which the player must then use in a series of challenges before they move on to the next tutorial. These challenges get progressively more complex, demonstrating the rationale for using the abstraction technique introduced in the next stage that will enable the player to produce a ‘better’ solution. Combining these two approaches means that a player is given structured guidance (explanations) but must then apply this knowledge in unguided challenge levels.

Motivation

The game motivates players to use the taught functionality through intrinsic integration (Habgood & Ainsworth, 2011) using block limits, collectable items, required block validation and obstacles (Aim 4). Each challenge limits the number of total blocks that can be used in the program, forcing the player to address block duplication and produce a nearly ideal solution. It also means that players cannot produce dead block code smells, because every block in the program must be used to achieve a maximum score on the level. Players must stop on each coin to collect it and must collect every coin to achieve a maximum score for that level (Section 7.4.4). Solutions are validated for containing the block related to that challenge. Some levels contain obstacles, such as enemy ghost ships, that will shoot at the player if they are within range. These can be avoided by hiding the ship using the ‘hide’ block. Levels contain rocks that will sink the ship if the player hits them. On the cloning challenges, there are sets of floating boxes that must be destroyed by cloning the cannonball sprite.

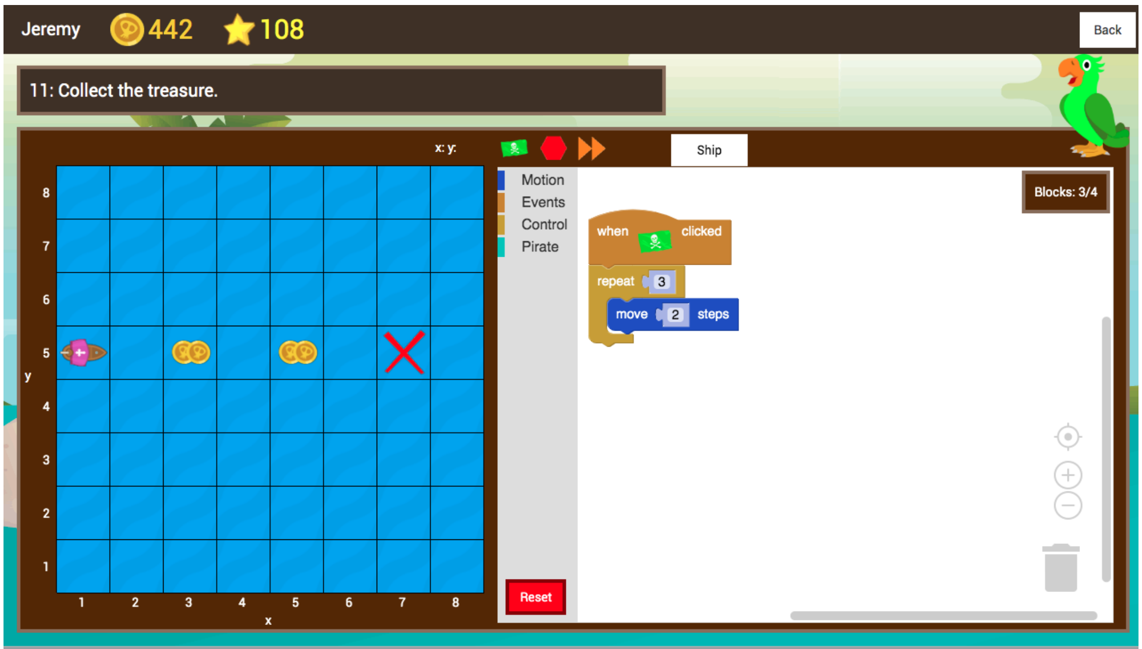
7.4.3 Scratch Integration

The Pirate Plunder layout and functionality are based on Scratch 2.0 (the most popular version used in schools at the time of development.) This addresses Aim 2, similarity to Scratch.

This section describes how the Pirate Plunder functionality and user interface was similar to (and differed from) Scratch, the programming blocks available in Pirate Plunder and how sprites were implemented.

Functionality

Pirate Plunder replicates the layout of Scratch 2.0 in that the scene is on the left and the program workspace is on the right (Figure 7.23). The buttons above the workspace work in the same way: the green flag executes any scripts attached to the ‘when green flag clicked’ block and the stop button stops the program execution. In Scratch, the currently executing script is highlighted (not individual block) because multiple scripts can be executed concurrently. In Pirate Plunder, execution has been slowed down to make the program easier to debug, with the current instruction highlighted to allow the player to better ‘trace’ the code (Section 2.2.2). Scripts cannot be executed concurrently in Pirate Plunder because of implementation limitations. There is a fast forward button that allows the player to speed up program execution, similar to Lightbot (Section 2.4.5).



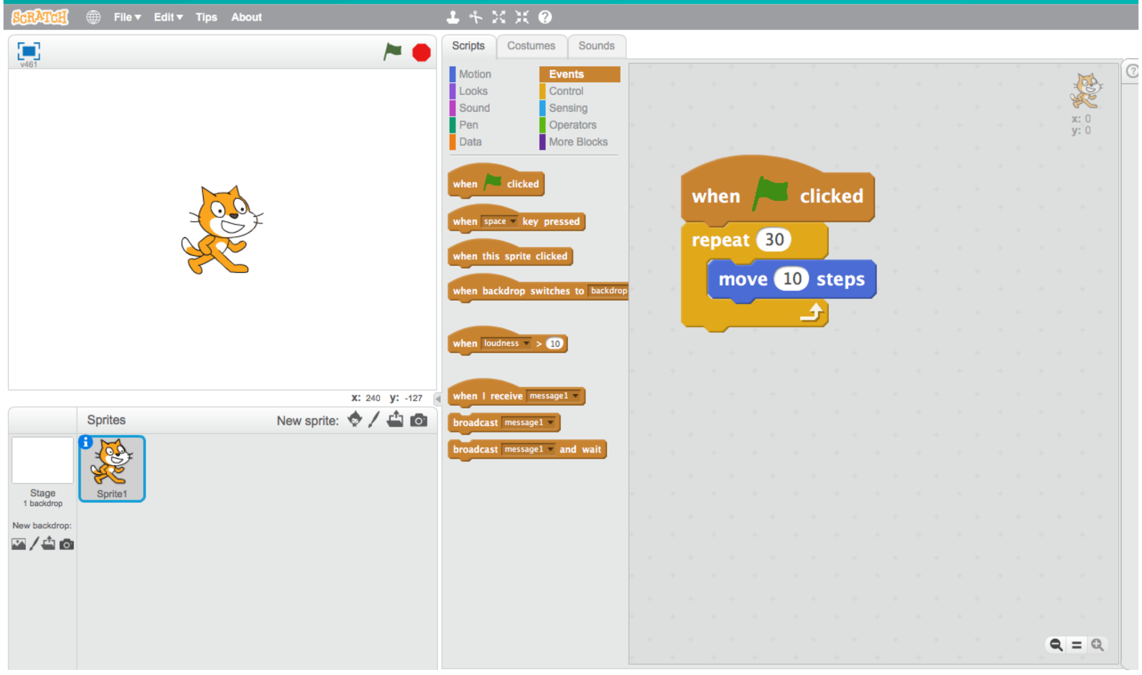


Figure 7.23: Screenshots of Pirate Plunder (top) and Scratch 2.0 (bottom)

The program workspace is implemented using Google Blockly, a library for creating block-based programming languages and editors that can be easily converted to text-based languages including JavaScript and Python. Code.org (Section 2.4.5) and Scratch 3.0 (the latest version at the time of writing) both use Blockly. Scratch 3.0 was in development during the Pirate Plunder design and development phase, hence why it was not used. Blockly does not look like Scratch 2.0 by default, so block colours have been changed to match Scratch, block highlighting on execution was switched from a bezel to a yellow outline and unattached blocks made identical to attached blocks (they are greyed out by default in Blockly.) Custom block and cloning functionality mimic Scratch as closely as possible. Program validation, restrictions and in-game warnings were added based on player feedback during testing (Section 7.4.6). Block names are validated (so the player cannot create two custom blocks with the same name), the player can only create ‘number’ inputs to avoid unnecessary complexity and the player cannot delete a custom block definition whilst that block is being used in the program.

The Pirate Plunder grid is top-down and uses an 8x8 grid in a single quadrant (top-right), compared to the Cartesian 480x360 pixel coordinate system used in Scratch (Figure 7.24). This is because four quadrant coordinates are not taught in the English national curriculum until Year 6 (age 10 and 11) (Department for Education, 2013). The coordinates for the grid refer to the grid spaces to make navigation and using the ‘go to position’ block easier for players. Like Scratch 2.0, Pirate Plunder has a coordinate position indicator that updates when the player hovers over the grid (shown in Figure 7.8).

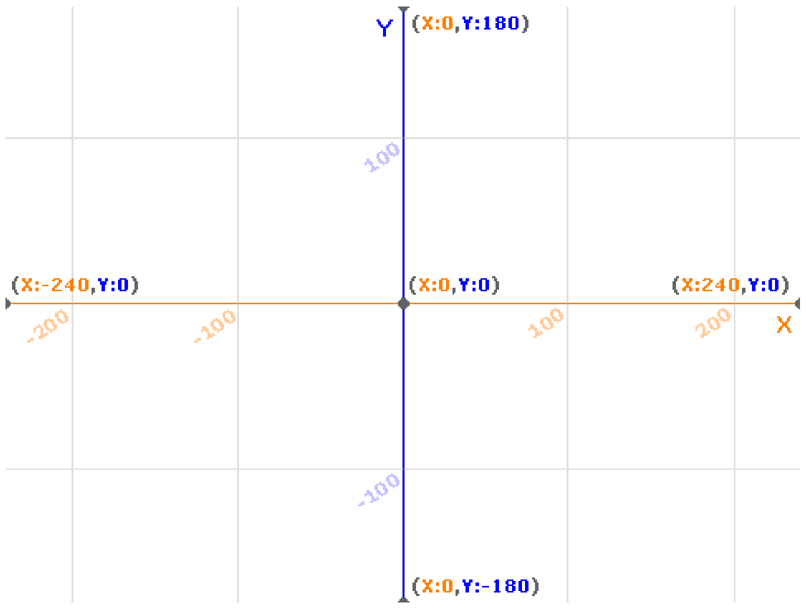


Figure 7.24: Screenshot of Scratch’s 480x360 four-quadrant pixel coordinate system, Pirate Plunder uses an 8x8 grid in a single quadrant (top-right)

Blocks

The Scratch 2.0 toolbox contains 116 blocks divided into 10 categories. These blocks are all available from the start to give the user freedom, in line with Scratch’s constructionist principles (Maloney et al., 2010). Yet, this can be both daunting and difficult for novice users. The large and always visible block pallet is one of the problems with Scratch described in Section 5.2.1 that may contribute to forming bad programming habits (Meerbaum-Salant et al., 2011). Pirate Plunder uses a restricted set of blocks relevant to gameplay that are introduced as the player progresses through the game (Table 7.2). The block categories are taken from Scratch (apart from the ‘Pirate’ category).

Table 7.2: Pirate Plunder programming blocks

Category	Block	Use in Pirate Plunder
Motion	Move	Move sprite *n* steps (grid spaces)
	Turn right	Turn sprite clockwise *n* degrees
	Turn left	Turn sprite anticlockwise *n* degrees
	Point in direction	Point sprite in a direction (up, down, left, right)
	Go to position	Move sprite to an *x*, *y* grid position
Looks	Show	Show sprite
	Hide	Hide sprite
Events	When green flag clicked	Execute a script when the green flag is clicked
Control	Repeat	Repeat the nested block(s) *n* times
	When I start as a clone	Execute a script when the sprite is cloned
	Create clone of	Create a clone of a sprite
	Delete this clone	Delete the clone of a sprite
Sensing	Property of	Get the *x* position, *y* position or direction of a sprite
More Blocks		Create and use custom blocks
Pirate	Get treasure!	Collect the treasure chest

Sprites

Scratch uses event-driven programming with multiple active ‘sprites’, where each sprite can be programmed separately. In Pirate Plunder, the available sprites are selectable above the program workspace, which is different from Scratch 2.0 where they are selectable in the bottom left corner (Figure 7.23). In Pirate Plunder, available sprites are level-dependent, and players cannot add, edit or remove them, unlike Scratch. The ship sprite is available in every level and the cannonball sprite is added in the cloning levels. Sprites face right at the start of each level and are visible by default. This restricts the player to use the taught concepts, whilst still providing similar functionality to Scratch (Aim 2).

7.4.4 Reward System

Pirate Plunder uses several reward strategies to motivate the player (Aim 4). These include the treasure mechanic, map coins and star ratings.

Treasure Chest

After collecting the treasure chest, the player receives a random number of coins between 1 and 15 (Figure 7.25). Ozcelik, Cagiltay & Ozcelik (2013) found that uncertainty enhanced learning in game-like environments and was associated with an increase in motivation. Collecting the treasure chest is the goal of every level apart from the first two tutorials before the ‘get treasure!’ block has been introduced.

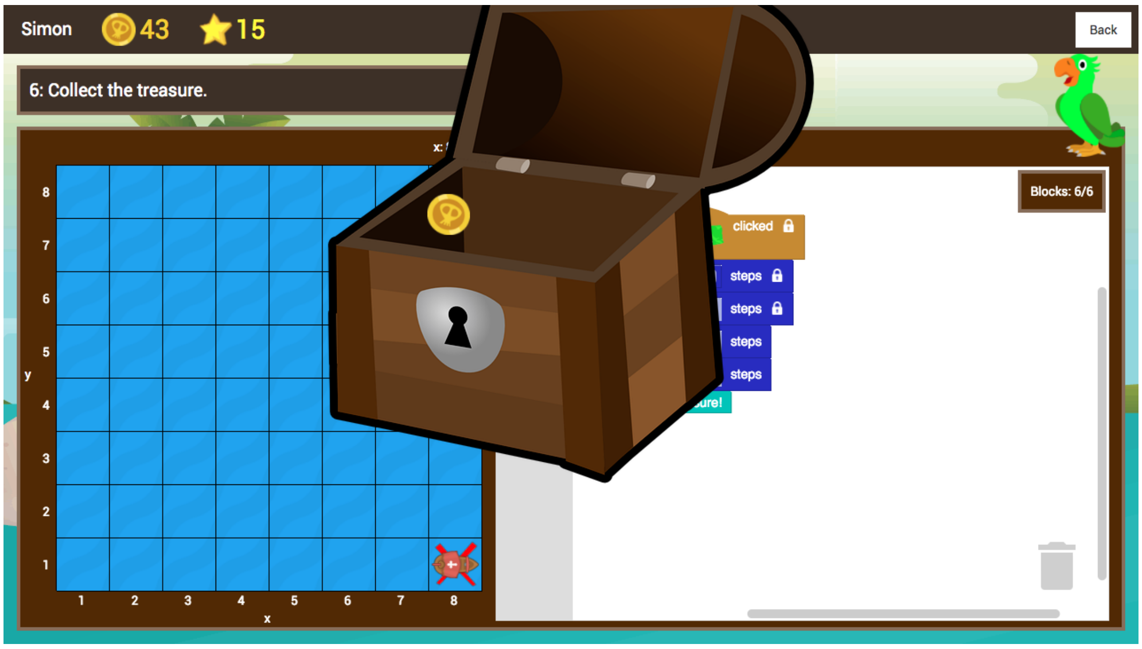


Figure 7.25: Screenshot of treasure chest collection in Pirate Plunder

Map Coins

Map coins are used in Pirate Plunder to get the player to stop the ship on each grid position. This is because in Scratch, repeating ‘move’ blocks is a method for animating movement (used in the bottom of Figure 7.23), justifying the use of loops. Collecting map coins is an important part of the game because it is used to demonstrate the justification for using loops, custom blocks and inputs (Section 7.4.1). Map coins come in denominations of either one, two or five coins per grid space to give the player extra motivation to collect them on some levels (the two-coin denomination is used in the top of Figure 7.23). The player can then use the coins collected from the map and the treasure chest to purchase items to customise their avatar (Section 7.4.7).

Star Rating

Players are given performance feedback (Malone & Lepper, 1987) through a star rating upon completion of each challenge (Figure 7.26). This is based on how many map coins the player has collected: three stars for all, two stars for some and one star for none. This is designed to further motivate the player to collect map coins, therefore completing levels using the taught concepts. This is based on similar three-star rating systems used in popular mobile games such as Angry Birds and Cut The Rope. The star count is shown for each level on the level select and the player's total is shown next to the coin count in the top bar of the game. It is also visible in the class screen (Section 7.4.7) so that the player can compare their star count with other players.

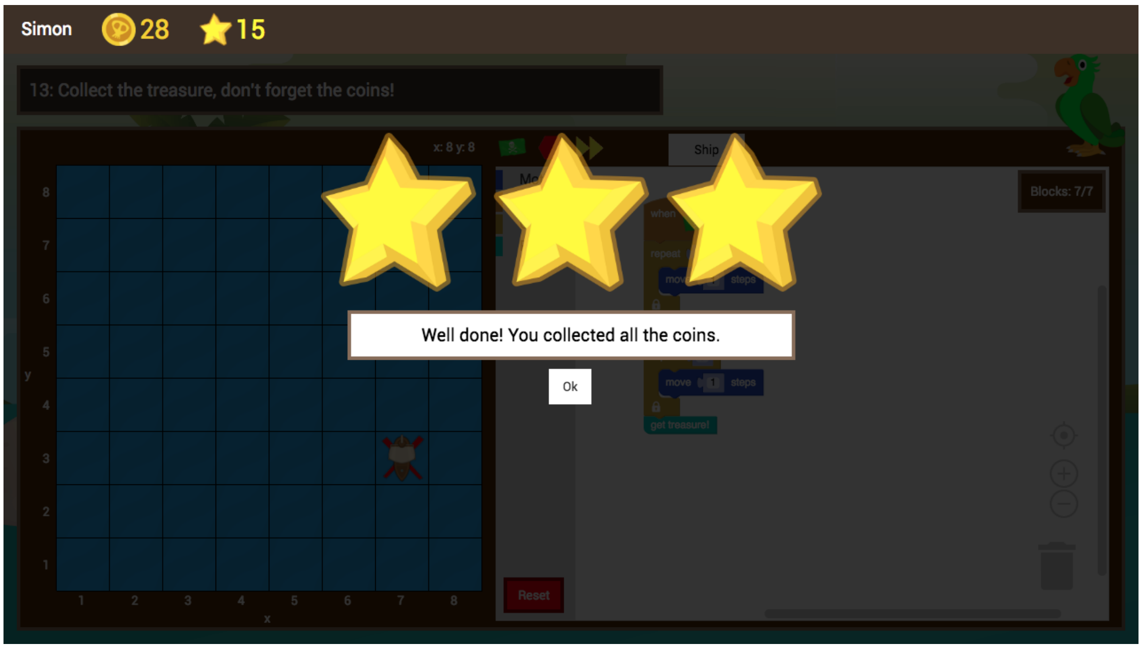

Simon
28
15
13: Collect the treasure, don't forget the coins!
x: 8 y: 8
Ship
Blocks: 7/7
Well done! You collected all the coins.
Ok
move 1 steps
get treasure!
Reset

Figure 7.26: Screenshot of star ratings in Pirate Plunder

7.4.5 Tutorials

As explained in Sections 7.4.1 and 7.4.2, tutorial levels introduce the player to blocks or functionality that they then use in a set of corresponding challenge levels. Tutorials differ from challenges in that they have a red parrot that gives explanations and instructions on how to use the block correctly (Figure 7.7). The parrot moves around the screen for each instruction and there is a pointer that demonstrates the action that the player needs to perform. In some tutorials, such a 'go to position', the player must demonstrate an additional competency such as understanding coordinates by clicking on specific grid positions (Figure 7.9). Players can only run the program once they have completed all the instructions. Tutorials are a key part of the game because of the difficulties that children have with understanding abstraction and are designed to require no additional classroom support (Aim 3).

This section explains the inspiration for the tutorial levels and how the instructions and pointer both work.

Inspiration

The tutorial levels were inspired by Dragon Architect and Stagecast Creator. In Dragon Architect (Figure 7.27), the player must complete basic tutorials before they can move to a sandbox level. They are then able to go back and 'learn' more functionality through additional tutorials, unlocking this in the sandbox. This 'guided-discovery' approach combines open-ended exploration with linear sets of puzzles (Bauer, Butler, & Popović, 2017). The tutorial levels contain a dragon character that moves around the screen to help direct the player (7.27A). However, this is on an automatic timer and can easily be ignored.

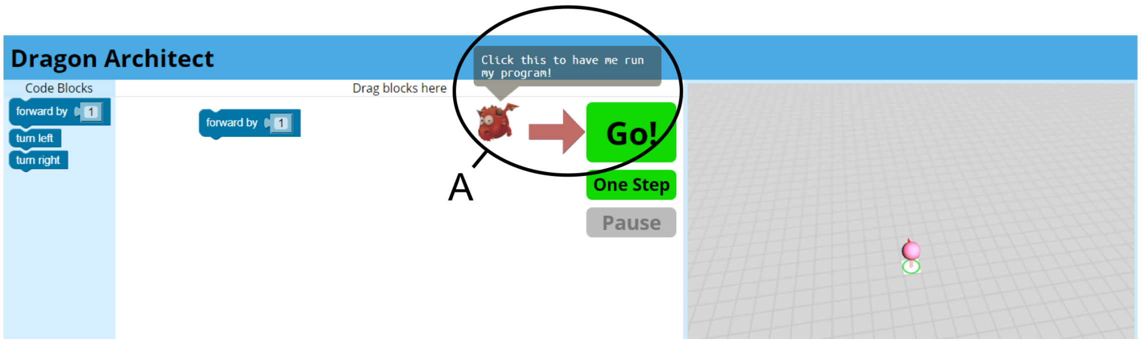
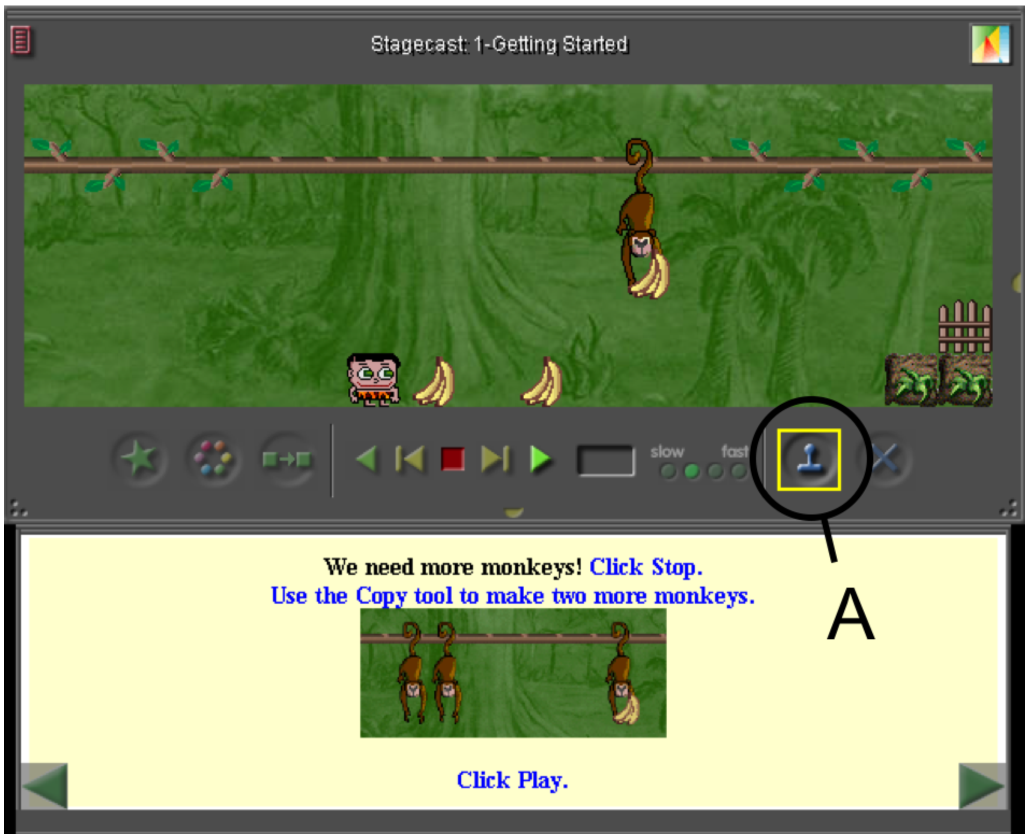


Figure 7.27: Screenshot of a tutorial in Dragon Architect, A) The help dragon that moves around the screen



We need more monkeys! Click Stop.
Use the Copy tool to make two more monkeys.

Click Play.

Figure 7.28: Screenshot of a tutorial in Stagecast Creator, A) Highlighting the required action

Stagecast Creator is a rule-based visual programming environment that can be used to create stories, animations and games (similar to Scratch). It has a 'Learn Creator' section that contains tutorials on how to use the application. These tutorials explain things like rules, mouse and keyboard interaction and variables. They work by having an explanation on the bottom of the screen, then restricting the application so that the player can only perform the next 'action' in the tutorial (Figure 7.28). These actions are highlighted in the application (7.28A). Stagecast Creator uses context and repetition to introduce concepts using minimal text.

Pirate Plunder uses a combination of these two approaches. Players must follow a series of actions within a tutorial that are highlighted on-screen by a moving help character.

Instructions

The instructions in Pirate Plunder tutorial levels are designed so that they can be completed without reading any text. It was observed in Study 1 (Chapter 4) that children would often ignore instructions and instead use a trial and error approach to figure out how the game worked (Section 4.4.2). Pirate Plunder tutorials have the player perform actions, which cannot be skipped or ignored like the instructions in Dragon Architect or Lightbot Jr, meaning that they cannot progress through the tutorial without completing them. Instructions are given by the red parrot, which moves around the screen depending on the instruction. The green parrot then gives feedback on whether the action has been done correctly or not (Section 7.4.6). The instructions can be cycled through by pressing the 'next' and 'previous' buttons, the 'next' button disappears when an action is required to progress. The pointer aids the player in performing the required action. Section 7.3 describes an example tutorial.

In the custom block and inputs tutorials (Section 7.4.1), certain buttons are disabled as part of the instructions (for example when the custom block creation window is opened) and validation is in place to ensure the player creates the correct custom block with the correct inputs. Feedback is given to ensure that the player understands that this is the case.

Pointer

The pointer helps the player understand the current instruction. This mainly points at sections of the screen but will also demonstrate actions when required. For example, clicking on a block category, clicking and dragging a block into the program (Figure 7.20B) or right-clicking on an input to duplicate it.

7.4.6 In-Game Feedback and Hints

Players are given feedback on both tutorials and challenges by the green feedback parrot (Figure 7.29A). This is designed to support the player when playing the game and perform the role of an instructor (Aim 3). Feedback is given for guidance, warnings or level requirements. Guidance feedback is for general block use and program issues such as missing the 'get treasure!' or 'when green flag clicked' blocks, using 'get treasure!' on a grid position where there is no treasure, going out of bounds (7.29B), hitting objects or getting hit by an enemy. Warnings are for behaviour that might break the game, including recursion (having a custom block inside its define block), using 'get treasure!' in a 'repeat', custom block validation (if that block already exists, if it is a game block (e.g. 'move'), if it is a block or input name is a JavaScript keyword that will break the execution of the program, etc.) Level requirement feedback is given for not using required blocks, hitting the block limit and trying to collect treasure when hidden. Feedback is either shown when the player performs an action, when they run the program (not allowing execution) or when the program is running. As mentioned in the previous section, players are also given feedback in tutorial levels for completing actions successfully or not. Feedback is also given in tutorials if the player tries to run the program before getting to the end of the instructions.

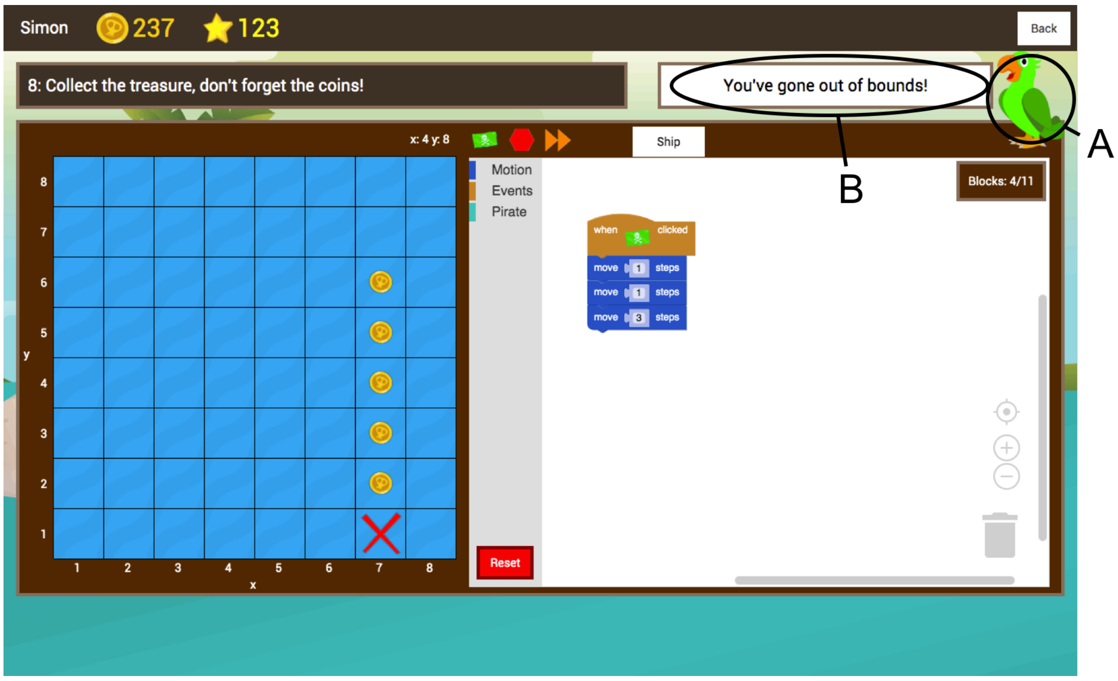


Figure 7.29: Screenshot showing an example of Pirate Plunder in-game feedback, telling the player that their ship has gone out of bounds, A) Feedback parrot, B) Feedback

Each challenge level has a 'hint' that the player can ask for, by clicking on the feedback parrot, which is designed to help them solve the level. This is known as micro-scaffolding (Melero, Hernández-Leo, & Blat, 2011) and can be used to help students when introducing new concepts in puzzle-based games.

7.4.7 Customisation

Players in Pirate Plunder have an avatar that they can customise in the shop. They can purchase items for their avatar or purchase new ships using the coins collected by playing levels, either map coins or those from treasure chests. This, along with collecting coins to buy items, is designed to motivate the player to keep playing the game (Aim 4). A strong link has been shown between self-designed avatars and game enjoyment, as players identify with and become invested in their character (Bailey, Wise, & Bolls, 2009). Player avatars and ships are integrated into the login screen and there is a 'class' screen where the player can compare their avatar with others. Players select their starting avatar from two options the first time that they play the game (Section 7.3).

This section explains the shop, class screen and how avatars are integrated into the Pirate Plunder login.

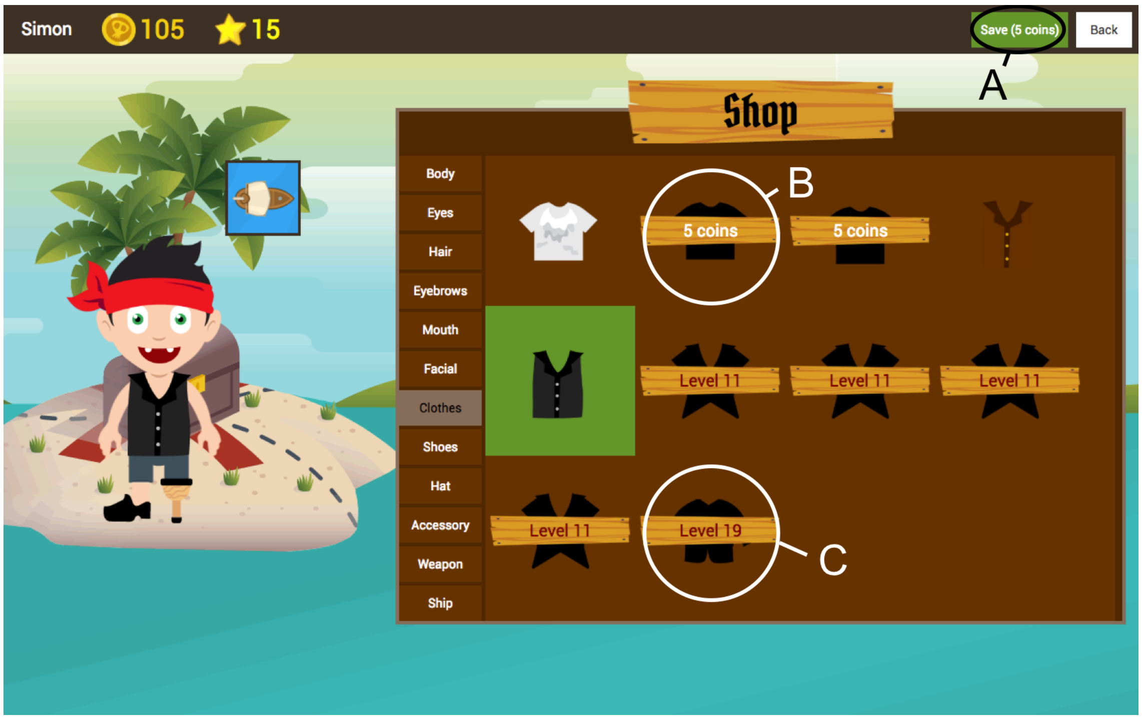


Figure 7.30: Screenshot of the Pirate Plunder shop, A) Paying five coins to save the updated avatar, B) Items that require purchasing, C) Locked items

Shop

The shop allows players to purchase items and customise their avatar (Figure 7.30). Items are split into 12 categories: body, eyes, hair, eyebrows, mouth, facial, clothes, shoes, hat, accessory, weapon and ship. Players are charged five coins each time they want to save their character (7.30A). This was done to limit customisation in the early levels, when the player has a limited number of coins, motivating them to play through early levels before spending time upgrading their character. Some shop items are available from the start, but most must be purchased (7.30B) and are unlocked after certain challenges have been completed (7.30C). A pricing model was used to price items based on the minimum and maximum coins a player could earn through tutorials and challenges. Additional shop items were added from player suggestions throughout development. The starting character and a few items were from a purchased graphics package, but additional shop items were created manually by the author.

Class

The class screen allows players to compare their avatars and statistics with other players (Figure 7.32). The class screen is not ranked by performance, but instead by the same ID order used on the login screen. This was done so to reduce the negative effects on intrinsic motivation that can arise from competition (Vellerand, Gauvin, & Halliwell, 1986), instead allowing players to compare their avatars and statistics without the pressure of a leaderboard.

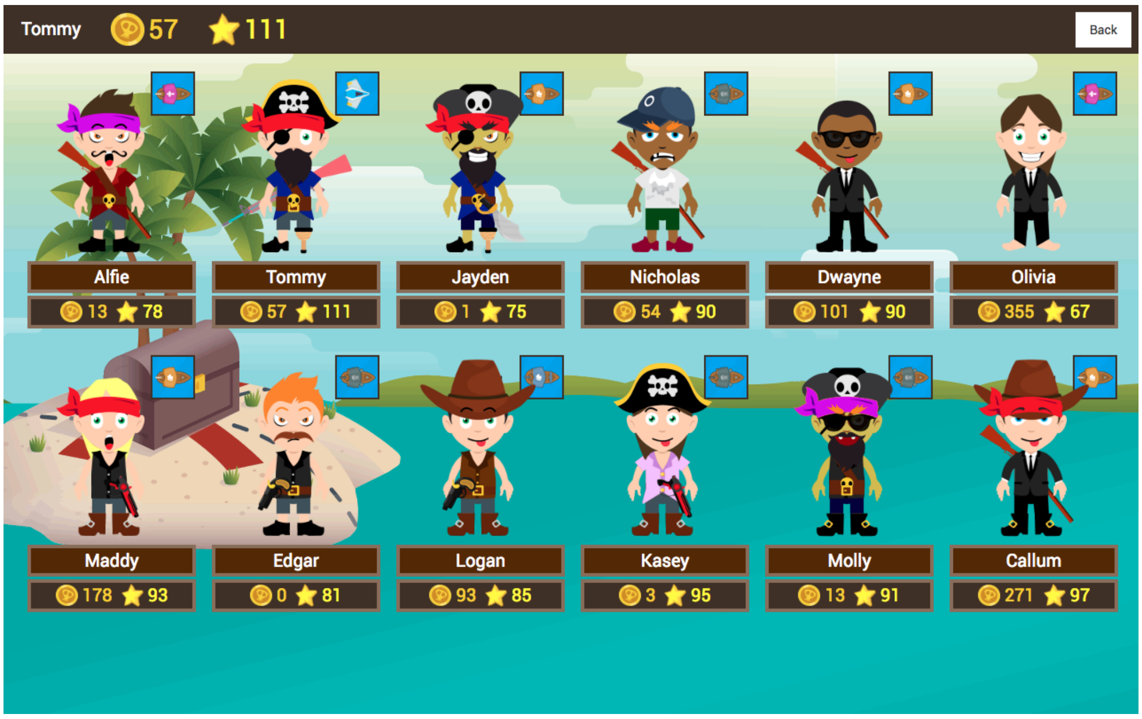


Figure 7.31: Screenshot of the Pirate Plunder class screen

Login

Player avatars are integrated into the Pirate Plunder login screen because they are not part of the gameplay. This makes them a more intrinsic part of the game and gives players an extra opportunity to compare avatars with their classmates. Players first enter their class ID, which then loads a selection screen similar to the right of Figure 7.32, listing each player's avatar, name and ship. A player then selects their character and is asked to enter a password. Once the player selects their avatar, it is then visible on every screen apart from the game itself.

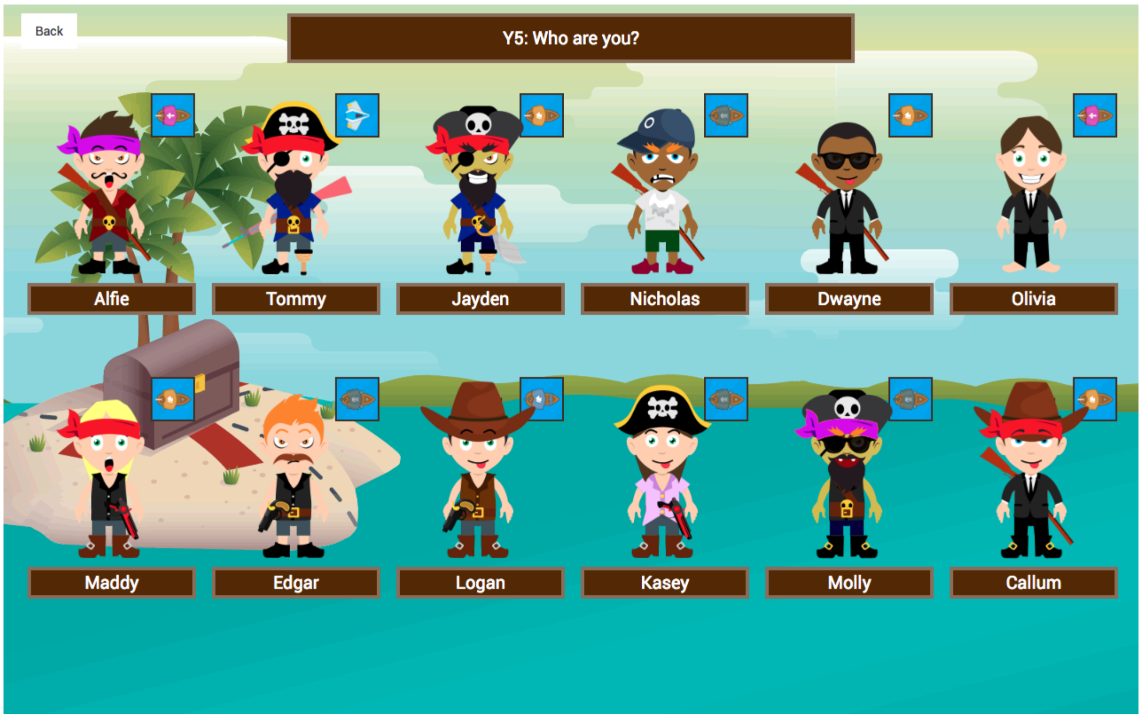


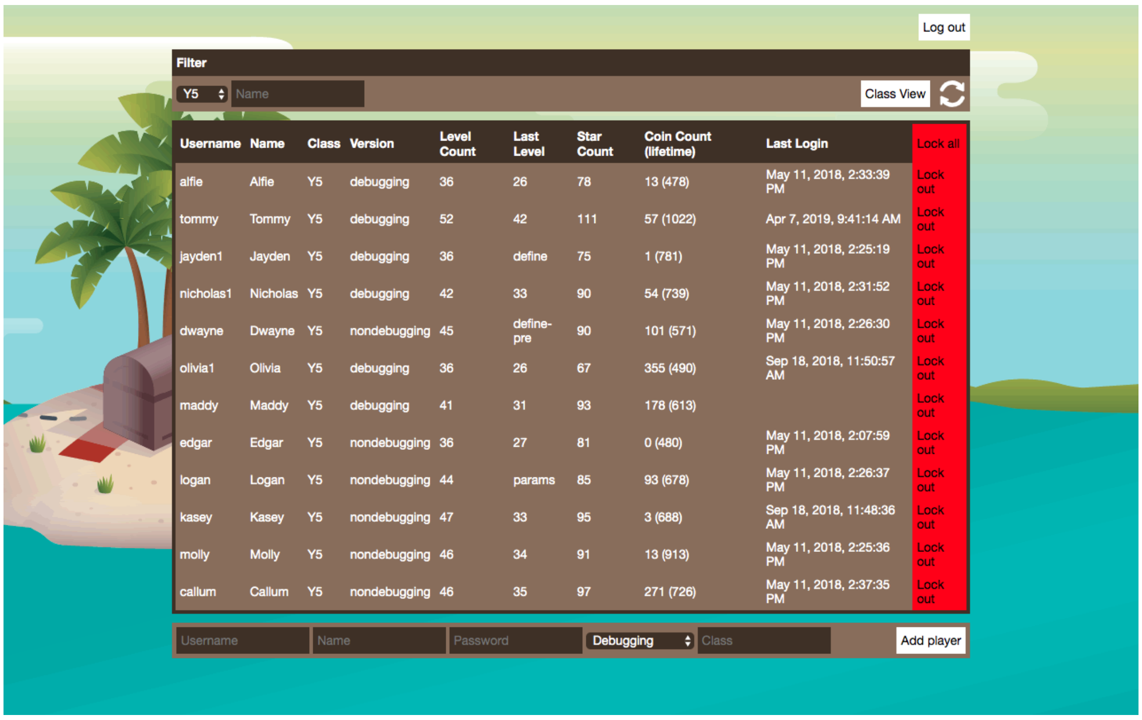
Figure 7.32: Screenshot showing part of the Pirate Plunder login screen

7.4.8 Administration Section

Pirate Plunder has an administration section that shows player information and statistics (Figure 7.33) (Aim 5). This also allows an administrator to lock players out, so they cannot use the game outside of the study sessions, and to reset passwords if required. It has a class filter and a search bar for player names. New player accounts can be created using the administration section, but in practice, a computer program was used to add them in bulk from a spreadsheet. In-depth statistics are shown for each player when selected. These are broken down into time spent on sections, shop purchases and level completions. The administration section can be used to track player progress and rank players on success metrics such as star count and levels completed. In this version, the first name of the player is saved into the database and a numeric user ID is used to save all other data. Name customisation was introduced for Study 4 (Chapter 9), meaning that the player's real name was not saved to the database or used in the game.

7.4.9 Analytics

Pirate Plunder saves analytics for player actions. These were used to investigate player approaches and performance in the game, particularly during development and between versions in Study 3 (Aim 5). They were collected for:



Username	Name	Class	Version	Level Count	Last Level	Star Count	Coin Count (lifetime)	Last Login	Lock all
alfie	Alfie	Y5	debugging	36	26	78	13 (478)	May 11, 2018, 2:33:39 PM	Lock out
tommy	Tommy	Y5	debugging	52	42	111	57 (1022)	Apr 7, 2019, 9:41:14 AM	Lock out
jayden1	Jayden	Y5	debugging	36	define	75	1 (781)	May 11, 2018, 2:25:19 PM	Lock out
nicholas1	Nicholas	Y5	debugging	42	33	90	54 (739)	May 11, 2018, 2:31:52 PM	Lock out
dwayne	Dwayne	Y5	nondebugging	45	define-pre	90	101 (571)	May 11, 2018, 2:26:30 PM	Lock out
olivia1	Olivia	Y5	debugging	36	26	67	355 (490)	Sep 18, 2018, 11:50:57 AM	Lock out
maddy	Maddy	Y5	debugging	41	31	93	178 (613)		Lock out
edgar	Edgar	Y5	nondebugging	36	27	81	0 (480)	May 11, 2018, 2:07:59 PM	Lock out
logan	Logan	Y5	nondebugging	44	params	85	93 (678)	May 11, 2018, 2:26:37 PM	Lock out
kasey	Kasey	Y5	nondebugging	47	33	95	3 (688)	Sep 18, 2018, 11:48:36 AM	Lock out
molly	Molly	Y5	nondebugging	46	34	91	13 (913)	May 11, 2018, 2:25:36 PM	Lock out
callum	Callum	Y5	nondebugging	46	35	97	271 (726)	May 11, 2018, 2:37:35 PM	Lock out

Figure 7.33: Screenshot of the Pirate Plunder administration section (table view)

- Changing game section (e.g. level select, shop, level) - used to calculate time spent on each section.
- Level attempt (current program, fast forward on, block count, errors).
- Level completion (time spent, stars collected, attempts, block count, hints used).
- Purchasing shop items.
- Working on the program (block creation, move and deletion).

7.4.10 Sounds

Sounds are used to give the player feedback, including completing a correct action on a tutorial, collecting coins, each star on a star rating, collecting treasure, firing a cannonball and when the ship was sunk.

7.5 Iterative Development Process

Pirate Plunder was created using an iterative development process that involved informal testing in three schools with small groups of children over the initial development period (up to Study 3). This process was split into the four stages shown in Table 7.3. Firstly, an early version of the game was tested informally with two children (age 7 and 9). Stage 2 then took place in an after-school club (with three children age 8 and 9) for roughly an hour. The third stage involved two children (age 10) playing Pirate

Plunder over three weeks after-school (the first week was with only one child so this is split into stages 3 and 3.1). The fourth stage was a pilot study comprising of 12 children (age 9 and 10) playing the game over four weeks (one session per week). The pilot study is discussed in more detail in the next chapter (Chapter 8).

Table 7.3: Pirate Plunder iterative testing stages

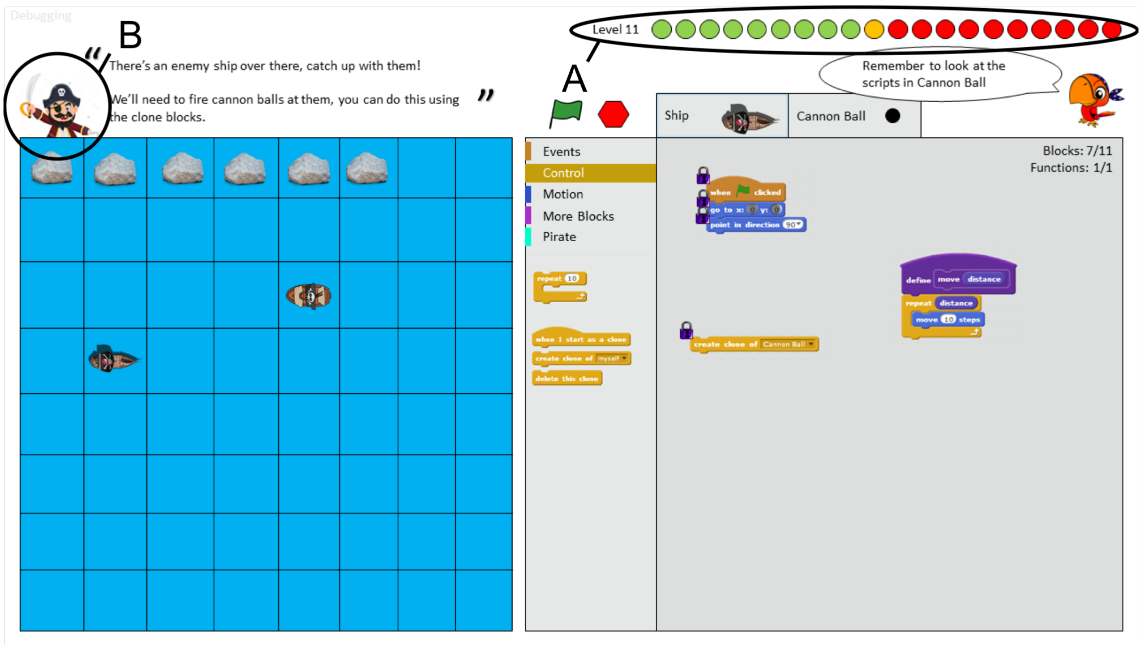
Stage	*N*	Age (Years)	Total Length
1	2	7 and 9	30 minutes
2	3	8 and 9	1 hour
3	1	10	1 hour
3.1	2	9 and 10	2 hours 30 minutes
4	12	9-10	5 hours

This section will describe some key developments at each stage of this iterative process. This process continued throughout Studies 3 and 4 with further changes discussed in Chapters 8 and 9.

7.5.1 Stage 1

Initial Idea

The initial idea for the game is shown in Figure 7.34. The player would play through a series of levels that would introduce custom blocks and cloning (7.34A), using Scratch blocks to move a pirate ship around a grid. They would be given instructions on each level by the captain of the ship (7.34B).



"There's an enemy ship over there, catch up with them!

We'll need to fire cannon balls at them, you can do this using the clone blocks."

Remember to look at the scripts in Cannon Ball

```
repeat 10

when I start as a clone
create clone of myself
delete this clone

when clicked
go to x: 0 y: 0
point in direction 90

create clone of Cannon Ball

define move distance
repeat distance
    move 10 steps
```

Figure 7.34: Screenshot of the first Pirate Plunder design, A) The game levels, B) Level instructions

Tutorial Order

An early version of the game is shown in Figure 7.35. At this point, the game introduced 'point in direction' after 'repeat'. This was altered before Stage 1 to give the player more levels and functionality to practice simple navigation before loops were introduced.

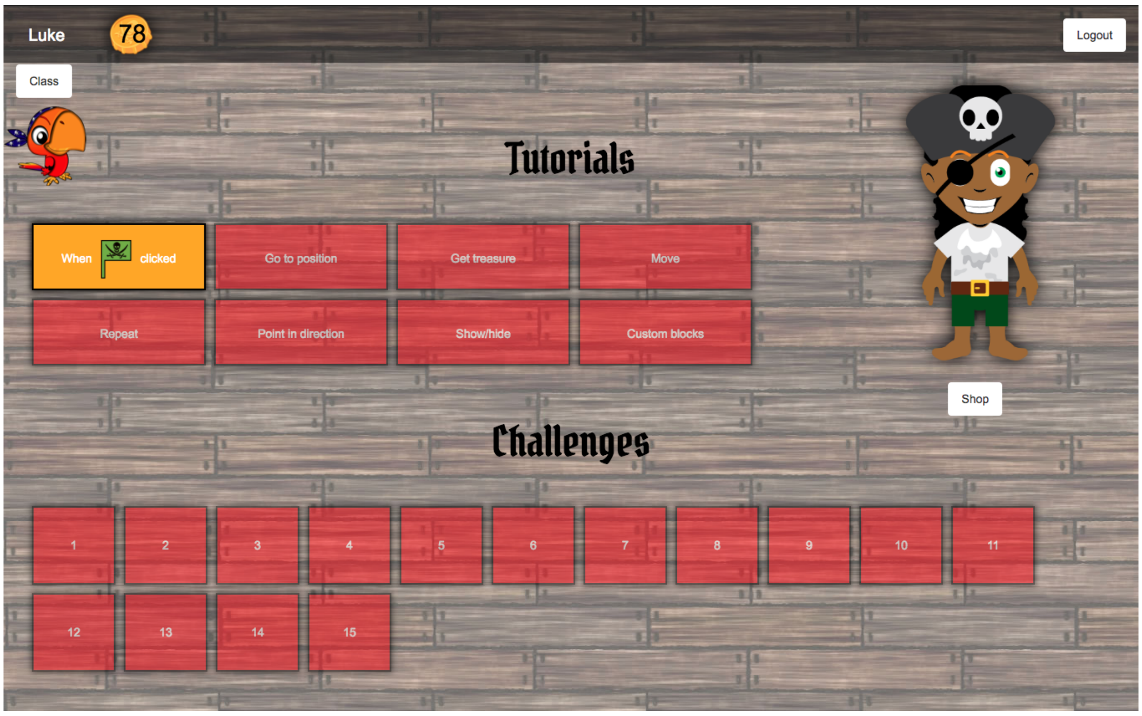


Figure 7.35: Screenshot of a Pirate Plunder tutorial at stage 1 of development

Skipping Tutorial Instructions

Tutorial instructions at Stage 1 were in a fixed position at the top of the screen (Figure 7.36). This meant that the 'next' button could be repeatedly clicked without the player reading any of the instructions (7.36A). It was observed during Stage 1 that players would do this and then get stuck because they did not know what to do. This resulted in the moving instructions, required actions and pointer described in Section 7.4.5.

Turning Sprites

Early versions of the game used the 'point in direction' Scratch block instead of 'turn right' and 'turn left' for basic navigation (Figure 7.36B). This came from Study 2 (Chapter 6), where the four Scratch projects use the 'point in direction' block to set the direction of the cat sprite. There was some confusion with the 'point in direction' input, which takes the number of degrees from the sprite's default direction (right). Some players struggled with understanding the number of degrees when it was passed into the custom block (e.g. Figure 6.3), because it loses the direction indicator ('left', 'right', 'up' or 'down'). 'Point in direction' was therefore switched to the turning. Clockwise

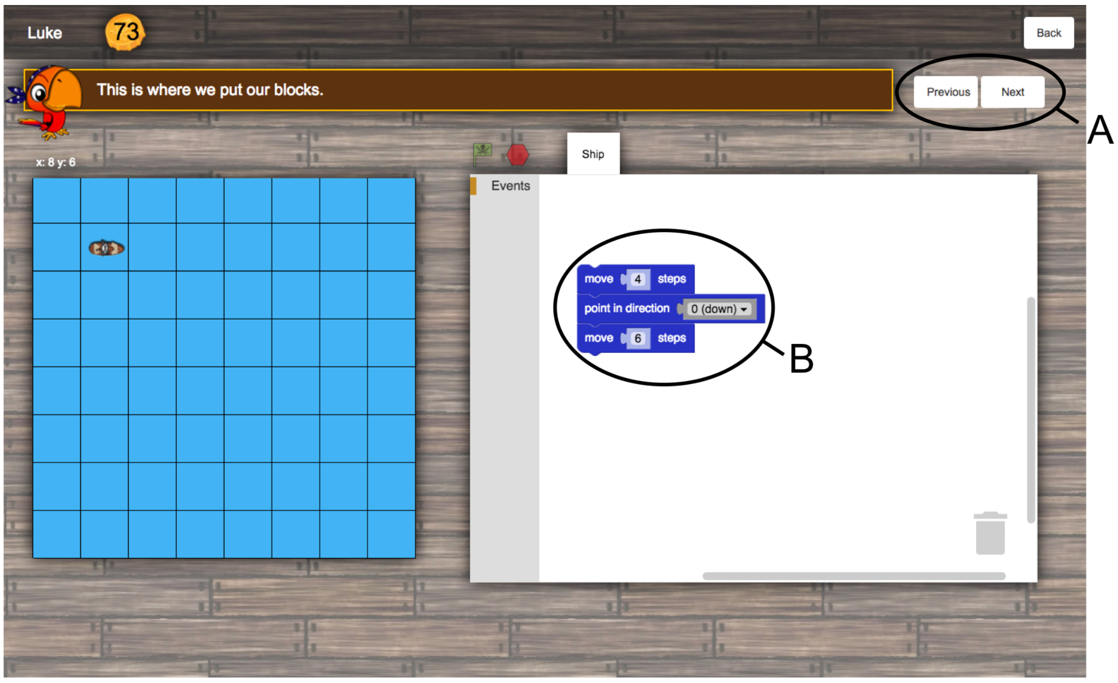


Figure 7.36: Screenshot of the Pirate Plunder level select at Stage 1 of development, A) ‘Previous’ and ‘Next’ buttons. B) ‘point in direction’ instead of ‘turn’

and counter-clockwise rotation by *n* degrees is introduced in Year 5 (age 9 and 10) in England (Department for Education, 2013). ‘Point in direction’ was left in the game for the cannonball sprite as the cloning levels require it for pointing the cannonball in the same direction as the ship (Figure 7.21).

Missed Coins

In early versions of Pirate Plunder, the player could not go back to a completed level and collect any coins that they had missed. This was changed during Stage 1 so that players could ‘recomplete’ levels to collect all the coins. Once all map coins had been collected they would show up as greyed out when the player revisited the level.

7.5.2 Stage 2

Final Design

The final design (used in the game screenshots in the rest of the chapter) was implemented for Stage 2 of testing. This included new assets and backgrounds to make the game more appealing to players.

Integrating Avatars

The login screen was initially a simple username and password form. Player avatars were integrated into the login screen for Stage 2 because they are not part of the gameplay (Section 7.4.7).

Demonstrating Actions

It was observed during Stage 2 that players would misinterpret tutorial instructions or get confused as to the action that they were meant to perform. This would then impact their understanding of the block that the tutorial was introducing. To address this, the pointer was changed from an arrow to demonstrating actions such as ‘click and drag’ (Section 7.4.5).

Saving Level Completions

Another feature added after Stage 2 was saving the solution that the player used to complete the level and loading this when they replayed the level. This meant that they could go back and see how they had solved a similar level, which was particularly beneficial between sessions (during the studies this gap was often a week).

7.5.3 Stage 3

Locked Level Progression

During Stages 1, 2 and 3, players could progress through tutorials one-by-one, unlocking the challenges for each tutorial when they had completed it. The idea behind this was that players could ‘discover’ additional functionality whilst playing through the challenges at their own pace (similar to Dragon Architect). However, in practice, players ignored the challenges and played through each tutorial level instead. They then had problems with the more difficult tutorials because they did not have enough experience playing the game to understand the functionality and rationale for using ‘repeat’ blocks, custom blocks and cloning.

For Stage 3.1, the level progression was fixed so that players had to complete the tutorial, then the set of challenges for that tutorial, before they could attempt the next tutorial. A prompt was added when the player unlocked the first challenge (after completing the ‘when green flag clicked’, ‘go to position’ and ‘get treasure’ tutorials) to make this clear. This was much more successful in getting players to understand and use the concepts correctly.

Custom Blocks and Inputs Tutorials

As it took around an hour of gameplay to reach the custom block levels, these were not tested until Stage 3. Players struggled to understand the rationale for using these blocks, so two extra tutorials were introduced using the project comparison strategy from Study 2 (Chapter 6). One that showed reducing duplication using the extract method with custom blocks, then another that showed further abstraction into a single custom block with an input. These are both described in detail in the level-by-level description (Section 7.4.1).

Performance Feedback

An early version of the star rating system was added for Stage 3 as an extra motivator for collecting map coins. This was an indicator on the level select screen for each level that showed whether all the map coins had been collected (Figure 7.37). However, it was unclear to players what this indicated, so it was replaced with the star rating system described in Section 7.4.4 for Stage 3.1.

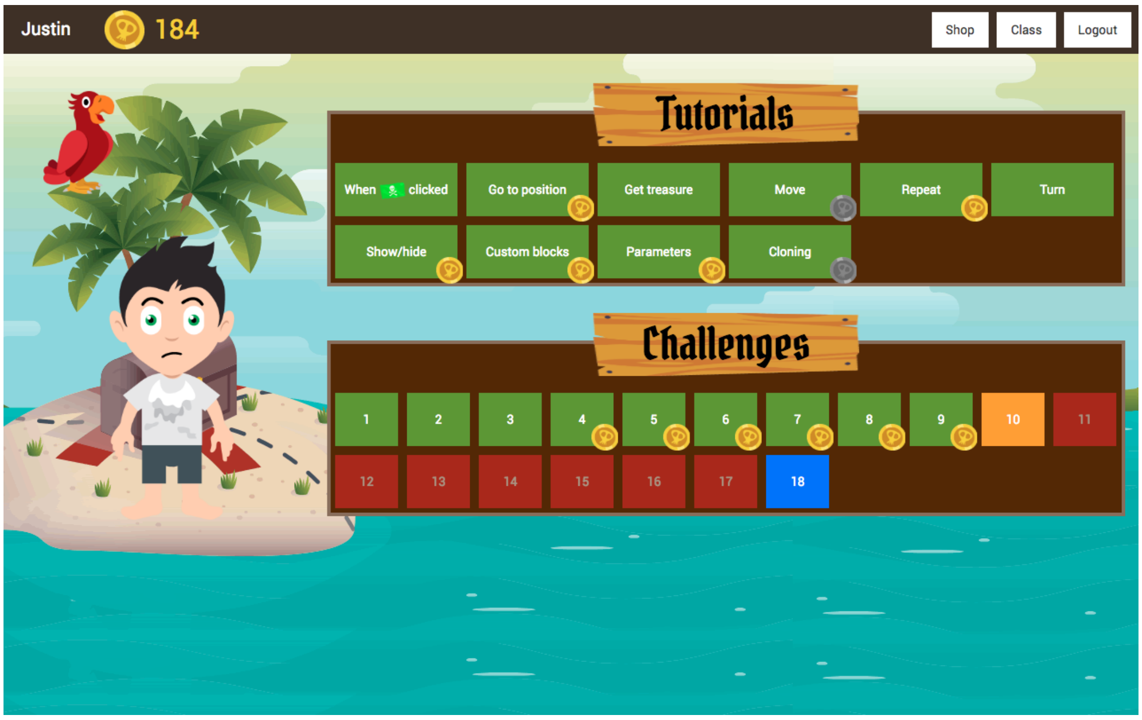


Figure 7.37: Screenshot of the early performance feedback in Pirate Plunder, a coin that indicates whether all the coins have been collected on that level

7.5.4 Stage 4

Level Attempts

For Stage 4, whenever the player goes 'back' from a level, they are asked whether they would like to save their level attempt. This allows them to attempt levels and either go back and check another solution or resume the level from the same point the next time they play the game.

Ship Customisation

Ship customisation was added during Stage 4 because players wanted to customise something that was part of the gameplay. They can purchase ships in the shop. These include pirate ships of different colours, rocket ships and a UFO.

Difficulty Progression

The Pirate Plunder version at Stage 4 had 45 levels (Table 7.4). This level progression was adjusted for Study 3 using game analytics collated level-by-level. It was observed that players were spending too long on levels before custom blocks were introduced and then having difficulties understanding them because the levels were too difficult. This resulted in the number of statement levels being reduced from 14 to 8, loops from 10 to 8 and cloning increased from 9 to 10. The level progression then contained 40 levels, with the player reaching loops at level 11, procedures at 19 and cloning at 31. The early custom block, inputs and cloning challenges were all simplified to help players understand them.

Table 7.4: How the Pirate Plunder difficulty progression changed between the pilot study and Study 3

Stage	Block (Tutorial)	Number of Challenges	
		Stage 4	Study 3
Statements	When green flag clicked Go to position Get treasure!	3	2
	Move	4	3
	Turn	7	5
Loops	Repeat	7	5
	Show/hide	3	3
Procedures	Custom blocks	6	6
	Inputs	6	6
Instances	Cloning (myself)		
	Cloning (other sprites)	9	10
Total		45	40

7.6 Summary

In summary, Pirate Plunder is designed to teach children to use custom blocks and cloning in Scratch. This chapter described elements of the game design and how these meet the aims of the game. These include difficulty progression, Scratch integration, reward system, tutorials, in-game feedback and hints, customisation, administration section, analytics and sounds. The game was developed over an iterative development process, which continues throughout the two studies described in the next two chapters (Chapters 8 and 9).

The next chapter (Chapter 8) describes an experimental study to investigate whether a debugging-first approach is beneficial for players. The study uses two versions of the game: debugging-first and non-debugging. These are compared against an active control group who were taught a standard Scratch curriculum. The results of this study feed into the final version of the game used in Study 4 (Chapter 9) to measure Pirate Plunder can be used to teach abstraction skills to primary school children.

Chapter 8

Study 3 - Investigating the Value of a Debugging-First Approach

Chapter 7 described the design and development of Pirate Plunder, a novel educational block-based programming game designed to teach children to use abstraction in Scratch. The rationale being that these abstraction skills should enable primary school children to recognise code smells in Scratch projects, such as duplicated blocks, and then be able to refactor their code using custom blocks and cloning to remove them. This should then help children avoid forming bad programming habits.

The study reported in this chapter aims to establish whether a debugging-first programming approach can benefit players. The results then feed back into the game design to produce a final version of Pirate Plunder for the study of its efficacy in the next chapter (Chapter 9). In this study, two versions of Pirate Plunder (debugging and non-debugging) were compared to a standard Scratch curriculum that does not introduce abstraction, with participants creating Scratch projects to a specification at pre-and post-test that would encourage code smells. The study also included an additional measure of computational thinking (CT) (Chapter 3), to see whether improvements in Scratch programming correspond to improvements in CT, as the literature indicates.

The chapter first explains the background of the study. It then describes a pilot study used to test Pirate Plunder and the assessment tasks, before moving onto the method, results, discussion and conclusions of the main study reported in this chapter.

8.1 Background

The background of the study is made up of three key subsections. The first subsection covers the main aim of Pirate Plunder: to teach children procedural abstraction skills (through custom blocks and cloning) that should enable them to refactor their code to remove duplication. The second subsection focuses on the debugging-first programming approach that differentiates the two versions of the game. The third subsection follows on from earlier chapters in examining the link between programming and CT in primary education, before going onto explain how it is applied in this study.

8.1.1 Abstraction and Code Smells

Abstraction and code smells are explained in detail in Chapter 5. Abstraction is an important part of programming. Good computer scientists can move easily between different levels of abstraction (Armoni, 2016). Lister (2011) suggests that novice programmers develop abstract reasoning skills as they gain expertise. Gibson (2012) argues that primary school children can acquire an understanding of abstract concepts, but only through structured teaching.

A code smell is a surface indication of an underlying problem in a program (M. Fowler, 1999). For example, duplicated code and long methods are both code smells that indicate bad program design. Code smells can be removed through refactoring: improving the internal structure of the program without altering its external behaviour. The extract method is one way of refactoring code to remove duplication code smells. It involves moving sections of code into their own procedure that can then be invoked from multiple places, a process known as procedural abstraction. This process is difficult for students at high-school and university level to understand, despite it being an important programming skill (Kallia & Sentance, 2017).

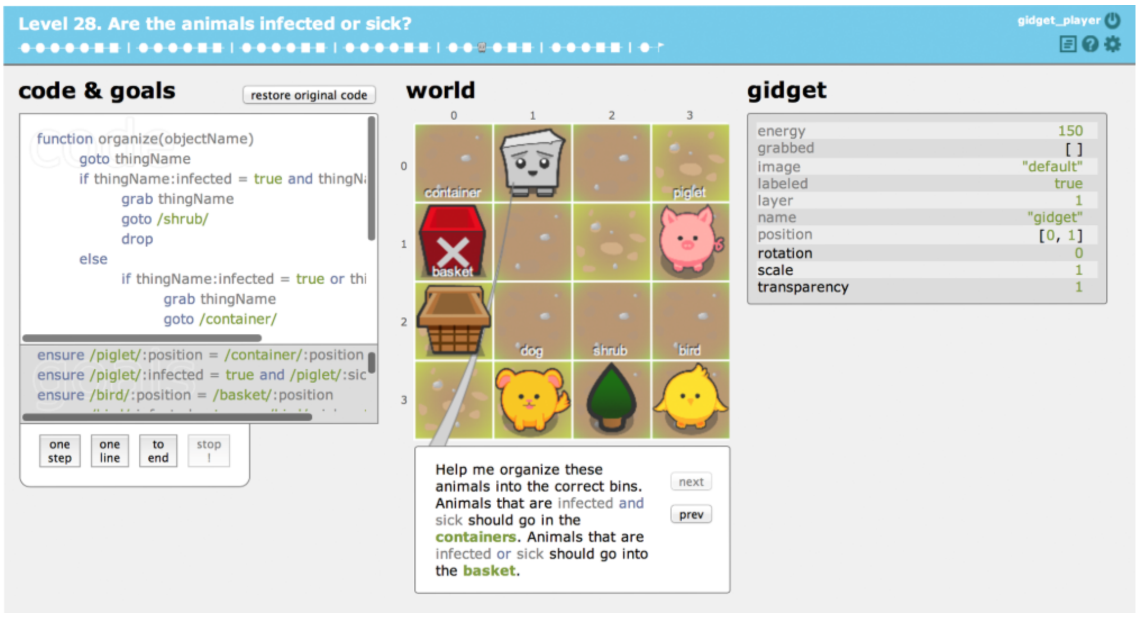
Duplicated code and long script smells are common in Scratch projects (Section 5.2.3) and can make them difficult to understand, debug and maintain (Aivaloglou & Hermans, 2016). Custom blocks and cloning can both be used to refactor code in Scratch. Custom blocks are the equivalent of procedures, allowing Scratch users to create their own blocks that can then be invoked from multiple places within a sprite (Section 5.3.2). Cloning allows the user to create copies of sprites, which can also be used to reduce sprite and block duplication (Section 5.3.3). Yet, these are rarely taught in Scratch curricula and even when they are used in Scratch projects, they do not reduce the amount of code duplication (Robles et al., 2017), suggesting that they are often not used correctly.

Dr. Scratch (Section 3.3.1) is a CT measure that scores projects across seven CT concepts based on the blocks used. One of these categories is abstraction and decomposition, which is measured by having multiple scripts in multiple sprites (1 point), using custom blocks (2 points) and using cloning (3 points). This provides an objective measure of abstraction in Scratch.

Pirate Plunder aims to teach primary school children to use custom blocks and cloning in Scratch through practical problem-solving. This is discussed in detail in Chapter 7. One of the aims of this study is to see whether children can apply the skills learnt in Pirate Plunder to Scratch projects. Yet, the main aim is to investigate the value of a debugging-first approach.

8.1.2 Debugging-First

The main aim of this study is to establish whether a debugging-first approach in Pirate Plunder benefits players when solving levels. The rationale behind using a debugging-first approach in programming tasks is discussed in Section 2.4.6. It comes from the theory that novice programmers learn better by completing existing code than by writing new code (Van Merriënboer & De Croock, 1992). This is known as the completion strategy, where part of the solution is visible so does not have to be held in working memory (Paas, 1992). Lee et al. (2014) list debugging-first as their first principle of debugging games, describing it as encouraging "learners to learn programming concepts by debugging existing programs before creating new programs... our approach provides nearly complete, but broken programs for learners to debug and fix before moving onto the more demanding task of creating new puzzles from scratch" (p. 57). This is the approach used in Lee's text-based programming puzzle game, Gidget, which showed promising results in getting novices to program using conditionals and loops (M. J. Lee & Ko, 2014) (Figure 8.1).


Level 28. Are the animals infected or sick?

code & goals

```
function organize(objectName)
    goto thingName
    if thingName:infected = true and thingN
        grab thingName
        goto /shrub/
        drop
    else
        if thingName:infected = true or thi
            grab thingName
            goto /container/
```

```
ensure /piglet/:position = /container/:position
ensure /piglet/:infected = true and /piglet/:sic
ensure /bird/:position = /basket/:position
```

gidget

energy	150
grabbed	[ ]
image	"default"
labeled	true
layer	1
name	"gidget"
position	[0, 1]
rotation	0
scale	1
transparency	1

Help me organize these animals into the correct bins. Animals that are infected and sick should go in the containers. Animals that are infected or sick should go into the basket.

Figure 8.1: Screenshot of a level from Gidget

The debugging-first version of Pirate Plunder (Figure 8.2) has blocks that either:

1. Have an incorrect input (8.2A).
2. Have a locked (but correct) input (8.2B).
3. Are there for assistance and are undeletable (8.2C).
4. Are there for assistance (but may not be needed) (8.2D).

The player must either use, change or remove these blocks to complete the level. Undeletable blocks have a white padlock and cannot be removed from the program.

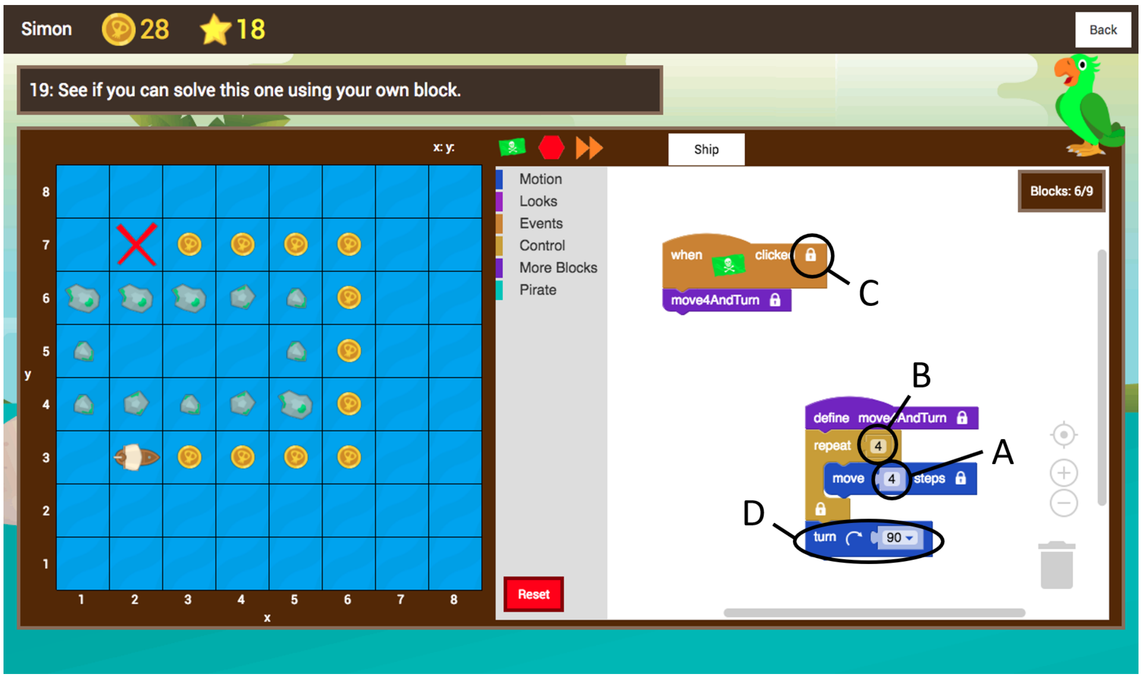


Figure 8.2: Screenshot of a debugging-first program in Pirate Plunder, A) An erroneous block that the player needs to delete (should be a ‘turn left’), B) A locked input that is the same colour as the block and cannot be changed, C) Padlock indicating the block is undeletable, D) An erroneous input that the player needs to change (should be ‘1’)

Locked inputs have the same background colour as the block they are in. The number and type of debugging-first blocks on each level is linked closely with the difficulty progression, with early challenges linked to a tutorial having more debugging blocks than later challenges (Appendix I). Figure 8.2 shows the first custom blocks challenge that has several debugging-first blocks. The debugging-first blocks are the only difference between the debugging-first and non-debugging versions of the game.

Antipatterns

Lee et al. (2014) observed several ‘antipatterns’ used by Gidget players that were counter-productive to problem-solving:

All knowing computer - Failing to scrutinise the debugging code, even if they are told it is filled with errors.

Reinventing the wheel - Deleting the debugging code without reading it and missing out on the clues that the code provides.

When you have a hammer, everything looks like a nail - Persisting in using programming constructs used for earlier levels.

I don’t want to try it - Avoiding trying out new ideas.

I'll use it as it is - Failing to adapt an existing example to a particular context (suggesting a lack of abstract reasoning that is common in novice programmers (Lister, 2011)).

The Pirate Plunder debugging-first version is designed to address these issues. By locking certain inputs and making blocks undeletable, players are restricted in 'reinventing the wheel'. This approach was inspired by Box Island (Radiant Games, 2016) (bottom right of Figure 2.8). The 'I don't want to try it' and 'when you have a hammer, everything looks like a nail' antipatterns are addressed by having conditions on level completion: block limits, obstacles, required blocks and the level progression itself where functionality is unlocked as the player progresses through the game. Debugging-first custom blocks and inputs are given erroneous names and large values in an attempt to address the 'I'll use it as it is' and 'all knowing computer' antipatterns.

This study aimed to measure, using pre-to post-test results, whether the debugging-first version of Pirate Plunder is more effective than a non-debugging version in meeting the learning outcomes: better Pirate Plunder performance, teaching children to use custom blocks and cloning in Scratch and improving CT. Game analytics were used to analyse differences in player approaches between the versions.

8.1.3 Computational Thinking

Chapter 3 explores CT, what it means for primary education and the concerns surrounding it. These concerns include how to assess CT and whether it can transfer to other subjects or skills. This has proven difficult, in part due to the wide range of different assessments available. Study 1 (Chapter 4) showed no difference between a programming game and phonics activities on story sequencing ability, supporting the results of much larger studies measuring CT improvements after a Scratch curriculum (Straw et al., 2017) and mathematics improvements after a Scratch curriculum (with CT as a secondary measure) (Boylan et al., 2018).

Section 4.5.1 charted a move away from CT onto measurable computer science outcomes. The last few chapters have concentrated on abstraction and code smells. Abstraction can be measured in Scratch using Dr. Scratch, a formative CT assessment tool. In this study, CT is reintroduced as a secondary measure, to see whether improvements in Scratch programming correspond to improvements in CT using a summative assessment. This study will use one of the more well researched measures, the Computational Thinking test (CTt) (Román-González et al., 2018a) (Section 3.3.3). The CTt is a multiple-choice assessment based on visual programming and has been used with children age 10 and 11 in other experimental trials (e.g. Brackmann et al., 2017).

8.2 Pilot Study

A pilot study was conducted to test Pirate Plunder and the assessment tasks. This took place in a medium-sized primary school in northern England. The participants were 12 children age 9 and 10, who were experienced in Scratch, having used it and other programming tools throughout their primary education (the school has strong links with educational technology companies), but were inexperienced with custom blocks and cloning. The school is below the national average (64%) with 54% of pupils meeting the expected standard in reading, writing and maths.

The study took place over five weeks (one session of 1 hour 30 minutes each week). In the first week, six participants did the Scratch task assessment (Section 8.3.3), before playing a debugging-first prototype of Pirate Plunder for the rest of the session. Six more participants joined them in the second week, playing the non-debugging prototype of the game. Observations made during this testing period and their impact on Pirate Plunder are discussed in Stage 4 of the iterative development process (Section 7.5.4). Participants then played Pirate Plunder for another three full sessions (five hours in total). In the last week, all the participants did the Scratch task assessment and the CTt.

The changes made to the materials and procedure from the pilot study are discussed in the next section.

8.3 Method

8.3.1 Participants

The participants in Study 3 were 85 children age 10 and 11 (*M* = 11.21, *SD* = 0.3) from a large primary school in northern England. The school is above the national average of pupils meeting the expected standard in reading, writing and maths with 87%. Participants were largely inexperienced with Scratch (having had sporadic lessons throughout primary school) and had no experience using custom blocks or cloning. The sample comprised of 36 males (42%) and 49 females (58%).

8.3.2 Design

The study followed a pre-test post-test quasi-experimental design to measure for differences between two versions of Pirate Plunder. The experiment consisted of three groups: Pirate Plunder (debugging-first), Pirate Plunder (non-debugging) and an active control group that followed a standard Scratch curriculum. Participants were assessed for their ability to use abstraction in Scratch using a Scratch task assessment and their computational thinking ability using the CTt. At post-test, they were given a Pirate Plunder or Scratch questionnaire depending on which group they were in. A selection of the Pirate Plunder participants were then interviewed after the post-test to establish whether they had understood the rationale for using abstraction.

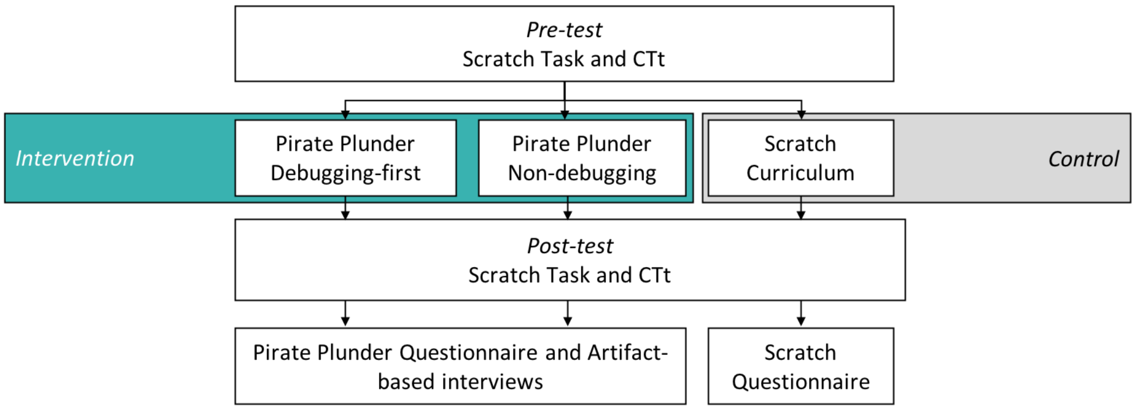

Pre-test
Scratch Task and CTt
Intervention
Pirate Plunder Debugging-first
Pirate Plunder Non-debugging
Scratch Curriculum
Control
Post-test
Scratch Task and CTt
Pirate Plunder Questionnaire and Artifact-based interviews
Scratch Questionnaire

Figure 8.3: Diagram of the Study 3 design

8.3.3 Materials

This section describes the intervention and active control group learning content, the quantitative assessments (Scratch task and CTt) and the qualitative data collection methods (questionnaires and artifact-based interviews).

Pirate Plunder

Pirate Plunder is described in detail in the previous chapter (Chapter 7). It is a novel educational block-based programming game designed for children age 9 to 11. It aims to teach code reuse using loops (repeat blocks), parameterised procedures (custom blocks) and instances (clones) in a game-based Scratch-like setting. Players program a pirate ship to navigate around a grid, collect items and interact with obstacles using Scratch blocks. They progress through a difficulty progression that forces them to duplicate code before introducing a block or strategy that they can then use to reduce duplication (Section 7.3). Block limits, collectable items, program validation and in-game feedback are all used to motivate the player. Pirate Plunder is designed to be played with minimal teacher interaction.

Two versions of Pirate Plunder were created for this study, a debugging-first version that starts with blocks in each level that the player must debug, and a version where the player starts each level with an empty program (Section 8.1.2).

Scratch: Animated Stories Curriculum

Study 1 (Chapter 4) highlighted the importance of using an active control group when measuring for cognitive improvements. An active control group aims to keep the experience of the control group similar to those during the intervention. This is done by giving them the same amount of researcher contact time and introducing them to similar ‘new experiences’ like using technology and learning content that they do not often use in school.

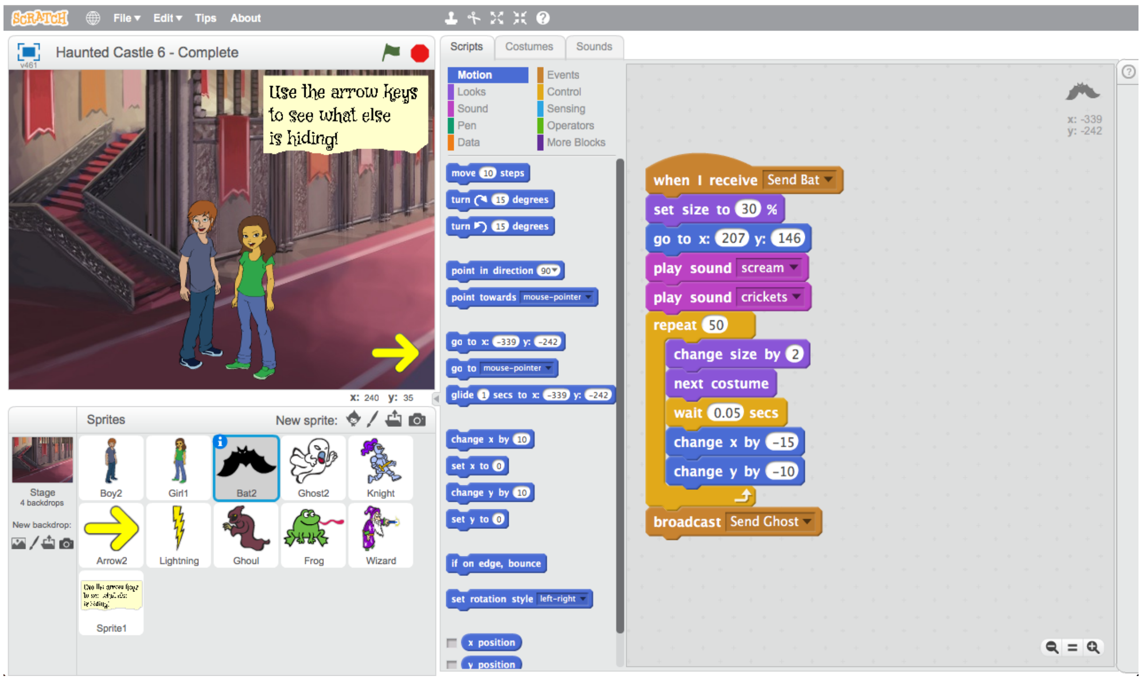
The control group in this study were taught a six-lesson Scratch curriculum produced by Twinkl, a popular educational resources company. The curriculum is designed for this age group (age 10 and 11) and does not cover custom blocks or cloning (Twinkl Educational Publishing, 2018). Table 8.1 gives a breakdown of each lesson. It was delivered by the author and involves creating an animated story based on a ‘haunted house’ (an example finished project is shown in Figure 8.4).

Table 8.1: Scratch: Animated Stories curriculum - lesson breakdown

Lesson Number	Lesson Name	Content	Scratch/Programming Concept
1	Animate a Scene	Animating characters to around a scene	Green flag events, sounds, repeats (loops), changing size, gliding to position
2	Broadcast a Message	Using message broadcasting (sending and receiving messages) to sequence events	Message broadcasting
3	Show and Hide	Using show and hide to set the visibility of sprites	Show and hide
4	Sequence a Story	Creating a story (using a storyboard) with different backdrops	Backdrops, speech
5	Adding Audio	Recording and adding audio to the project	Sounds
6	Getting interactive	Using key press events to add extra functionality	Key press events

Scratch Task

The Scratch task was completed at both pre-and post-test. It was designed to allow participants to demonstrate Scratch proficiency (as a baseline), but also to a specification that involved duplication, enabling them to use custom blocks and cloning if they were able to recognise that abstraction would be useful. The task was based on the Scratch projects used in Study 2 (Chapter 6) and involved animating the cat sprite around the edges of a rectangle, leaving an object on each corner. Two versions were used alternately (in an attempt to reduce copying), so two participants sat next to each other would be doing slightly different tasks. They were then given the other task at post-test. The tasks were different only in their appearance, having the participant place either trees or speakers on different rectangular backdrops (Appendices J and K). A third version was shown on the interactive whiteboard at the front of the classroom to demonstrate the expected behaviour. The project starts with the cat sprite



```
when I receive Send Bat
set size to 30 %
go to x: 207 y: 146
play sound scream
play sound crickets
repeat 50
    change size by 2
    next costume
    wait 0.05 secs
    change x by -15
    change y by -10
broadcast Send Ghost
```

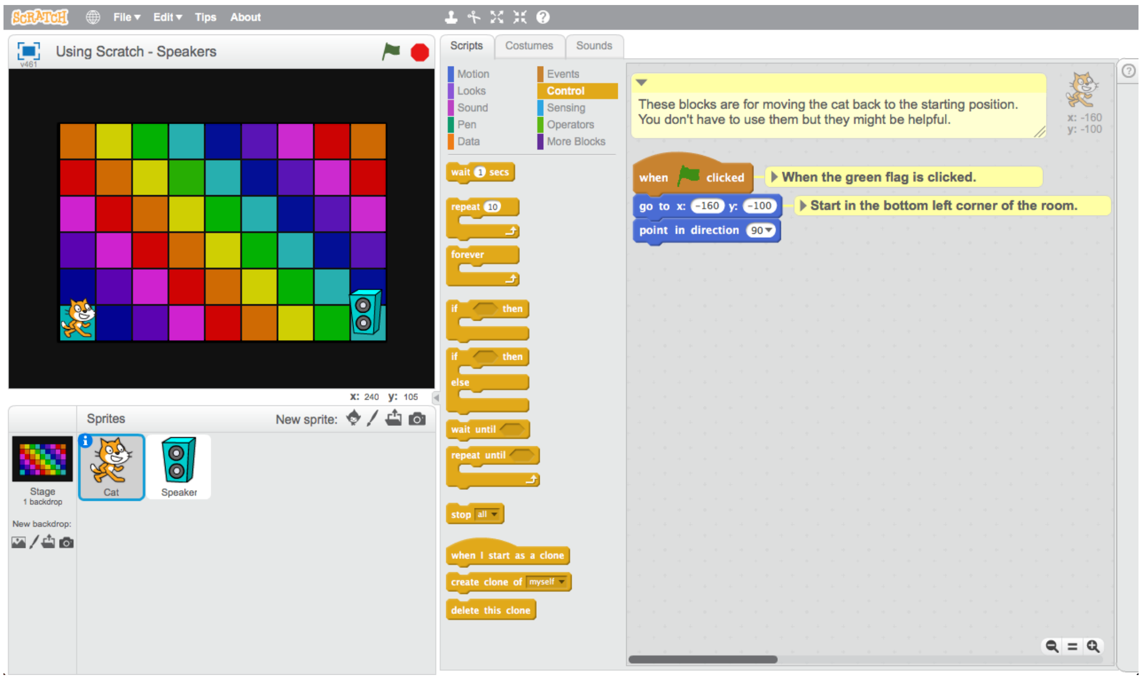
Figure 8.4: Screenshot of a finished project from lesson six of the Scratch curriculum

(containing a ‘when green flag clicked’ block, a ‘go to position’ block to move the cat back to the bottom left corner and a ‘point in direction’ block to reset its direction to facing right), a single empty object sprite (either a tree or a speaker) and a backdrop with a rectangle that the player must navigate around (Figure 8.5).

This version of the Scratch task was adjusted from observations during the pilot study. Comments were added to the starting instructions to explain what they did, and the task sheet was altered so that the expected behaviour was clearer.

The ideal solution to the setting up speakers Scratch task is shown in Figure 8.6. It uses a custom block for moving and turning different distances along the sides of the rectangle and clones a speaker on each corner. This block is used four times, one for each side of the rectangle. The speaker sprite is hidden when the green flag is clicked and moved to the position of the cat before being shown. This solution is similar to the cloning levels in Pirate Plunder (Section 7.4.1). Other ways to meet the specification (but not custom blocks and cloning) are similar to those used in Study 2 (Chapter 6). For example, having four object sprites, hiding them on ‘when green flag clicked’ and then showing them after the number of seconds that correspond to the cat sprite reaching that corner (e.g. Figure 6.2).

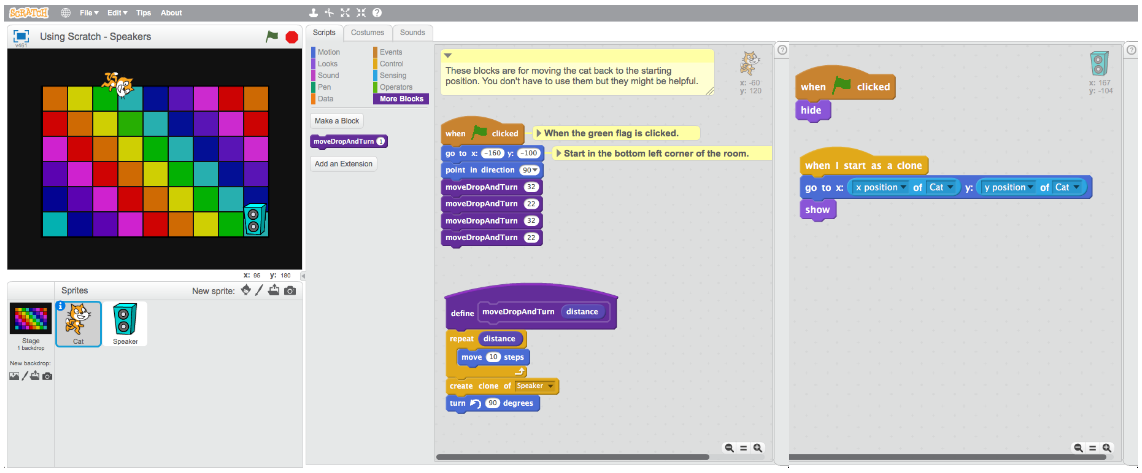
The Scratch task assessment projects were marked using Dr. Scratch for abstraction and decomposition (use of custom blocks and cloning) (Table 8.2) and for whether the project had met the expected outcome of the task. The expected outcome mark was split into cat behaviour and object behaviour (Table 8.3).



These blocks are for moving the cat back to the starting position. You don't have to use them but they might be helpful.

```
when green flag clicked          ▸ When the green flag is clicked.
go to x: -160 y: -100            ▸ Start in the bottom left corner of the room.
point in direction 90
```

Figure 8.5: Screenshot of the starter Scratch project for the setting up speakers task



These blocks are for moving the cat back to the starting position. You don't have to use them but they might be helpful.

```
when green flag clicked          ▸ When the green flag is clicked.
go to x: -160 y: -100            ▸ Start in the bottom left corner of the room.
point in direction 90
moveDropAndTurn 32
moveDropAndTurn 22
moveDropAndTurn 32
moveDropAndTurn 22

define moveDropAndTurn (distance)
repeat (distance)
    move 10 steps
create clone of Speaker
turn ↺ 90 degrees
```

```
when green flag clicked
hide

when I start as a clone
go to x: (x position of Cat) y: (y position of Cat)
show
```

Figure 8.6: Screenshot of a Scratch project showing the ideal solution to the setting up speakers task, showing the cat sprite (left) and speaker sprite (right)

Table 8.2: Dr. Scratch scoring system for abstraction and decomposition

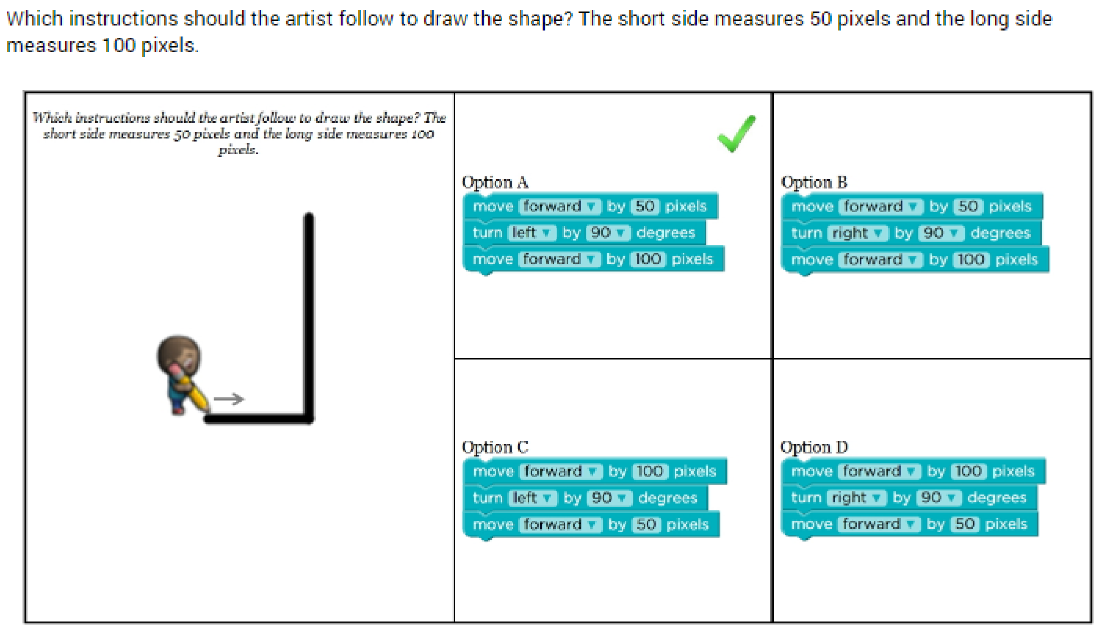
Points	Required Functionality
1	More than one script and more than one sprite
2	Custom blocks
3	Cloning

Table 8.3: Scratch task expected outcome marking (a score out of 2 for each category)

Points	Cat	Object
0	No movement	None/incomplete
1	Movement around the rectangle with no animation	Trees on each corner but not appearing when the Cat reaches them
2	Animated movement around the rectangle	Trees on each corner that appear when the Cat reaches them

Computational Thinking Test

The Computational Thinking test (CTt) is described in Section 3.3.3. It aims to measure "the ability to formulate and solve problems by relying on the fundamental concepts of computing, and using logic-syntax of programming languages: basic sequences, loops, iteration, conditionals, functions and variables" (Román-González et al., 2016, p. 4). The CTt contains 28 multiple choice questions that use visual arrows or blocks common in educational programming tools (Figure 8.7). Each question has four possible answers (with one correct) and not all questions have to be answered. It has been used in several experimental trials with children age 10 to 14 (e.g. Brackmann et al., 2017; Pérez-Marín et al., 2018) and has been used to predict performance on a Code.org course (Román-González et al., 2018a), suggesting that it may have some predictive validity in terms of 'computational talent'.



Which instructions should the artist follow to draw the shape? The short side measures 50 pixels and the long side measures 100 pixels.

Which instructions should the artist follow to draw the shape? The short side measures 50 pixels and the long side measures 100 pixels.

Option A ✓
```
move forward by 50 pixels
turn left by 90 degrees
move forward by 100 pixels
```

Option B
```
move forward by 50 pixels
turn right by 90 degrees
move forward by 100 pixels
```

Option C
```
move forward by 100 pixels
turn left by 90 degrees
move forward by 50 pixels
```

Option D
```
move forward by 100 pixels
turn right by 90 degrees
move forward by 50 pixels
```

Figure 8.7: Question from the Computational Thinking test

Questionnaires

Participants were given a questionnaire designed by the author at post-test. These were slightly different for the Pirate Plunder and Scratch groups (Appendices M and N) and contained the following questions:

1. How confident do you feel using Scratch?
2. How has your confidence using Scratch changed after playing Pirate Plunder/the Scratch lessons?
3. How confident do you feel using custom blocks in Scratch?
4. How has your confidence using custom blocks changed after playing Pirate Plunder/the Scratch lessons?
5. How confident do you feel using clones in Scratch?
6. How has your confidence using clones changed after playing Pirate Plunder/the Scratch lessons?
7. What did you like about Pirate Plunder/the Scratch lessons?
8. Is there anything that you would change about Pirate Plunder? (problems, difficulties, extra features)/the Scratch lessons?
9. Do you have any other comments?

The confidence questions (1, 3 and 5) used the scale: very confident, confident, slightly confident and not confident and answers were coded as quantitative data as 3, 2, 1, 0. The confidence change questions (2, 4 and 6) used a different scale: improved, same as before and declined and were coded as 1, 0, -1. The last three questions (7, 8 and 9) on the questionnaire were text boxes and optional. The coded answers were used as secondary data as self-reporting measures, particularly from children, can be inconsistent (Austin, Deary, Gibson, McGregor, & Dent, 1998). The feedback obtained from the last three questions on the Pirate Plunder questionnaire was used to improve the game for Study 4 (Section 8.5.4).

Artifact-Based Interviews

Artifact-based interviews are part of Brennan & Resnick's (2012) method of assessing the development of computational thinking through design activities in Scratch, along with project portfolio analysis and design scenarios. Their approach uses these interviews to discuss Scratch more generally, focusing on the user's background, project creation, involvement in the Scratch online community and wider interests.

In this study, the interviews were used to establish whether participants had understood the rationale for using abstraction through custom blocks and cloning. The interviews were one-to-one with the researcher. They began with open questions about the participant's project, to see if they could rationalise why they had done something without a prompt, before progressing to more leading questions about custom blocks and cloning. They were asked about their project (what each of the blocks did and why they had used them), alternative approaches they considered, why they had/had not used custom blocks, why they had/had not used cloning, before finishing on similarities between the task and Pirate Plunder and general feedback on the game. A full script can be seen in Appendix L.

8.3.4 Procedure

All participants did both the Scratch task and the CTt at pre-test. The Scratch task took place in the school IT suite in class groups. Participants were introduced to the study and the assessment, then given 40 minutes to produce a solution. This was done in order for the assessments to be conducted in one lesson (50 minutes) due to the time constraints of the study. They were told they could save multiple versions of their projects if they could think of more than to complete the task. The CTt took place after the Scratch task in the classroom on tablets. Participants were given 45 minutes to complete the test.

The class groups were then assigned to either the debugging-first intervention, non-debugging intervention or the Scratch control. School limitations meant that participants were placed in their class groups for the study. Sessions were conducted in their normal computing lesson time in the school IT suite. This meant that each group had 50 minutes per week during the intervention over six weeks (five hours of lesson time in total). The pre-tests and post-tests were conducted in the weeks before and after this period, making the study eight weeks long.

The Pirate Plunder sessions were unstructured, with participants playing through the game over six lessons. They were reminded not to copy from or complete levels for others. If they were stuck, the researcher or teacher would come around and give them hints on how to complete that level. The Scratch lessons followed the curriculum detailed in Section 8.3.3, one lesson per week.

At post-test, all the participants then did the Scratch task (following the alternate specification) and the CTt again following the same procedure, and in the same order, as the pre-test. After this, the participants were asked to complete either the Pirate Plunder or Scratch questionnaire and a sample of participants from the intervention groups were interviewed. The interviewees were selected based on their use of custom blocks, cloning or obvious duplication in their Scratch projects, roughly five participants per category. As not many participants used custom blocks or clones, all participants that did use them were interviewed, along with a set of participants who could have used abstraction but did not (having duplicated blocks that could be moved to a single custom block). The interviews were between 10 to 15 minutes and conducted in a reading room adjoining the classroom. They took place over 1 day (allowing time for 16 participants) as the end of the study coincided with the end of the school year.

The study took place in the summer term after participants had completed their Key Stage 2 SATs (Standard Assessment Tests). It was eight weeks long, a six-week intervention with assessments taking place in the weeks either side, one session per week. It ran from the 24th May 2018 to the 19th July 2018.

8.3.5 Ethics and Access to Participants

The ethics application for testing Pirate Plunder in primary schools (Appendix G) was approved by the University ethics committee as an amendment to the initial ethics application (Appendix C). This covers the game testing in Chapter 7 and Studies 3 and 4 with their respective pilots. For this study, permission was given by the headteacher to confirm that the study could go ahead. Once the teachers had been briefed, opt-out consent forms were sent to the parents/guardians of participants (Appendix O). This was done to meet the wishes of the school.

All data, including assessment scores, game analytics, coded questionnaires and interview transcripts, were anonymised and stored against participant ID numbers. Scratch projects were renamed as versions (e.g. ‘V1’, ‘V2’) and saved in participant ID folders. Appendix H shows the data management plan.

8.3.6 Hypotheses

There were two hypotheses for the study, the first based on evaluating the debugging-first approach and the second on the wider aims of Pirate Plunder:

1. The debugging-first version of Pirate Plunder would have a positive impact on learning outcomes (Dr. Scratch abstraction and CTt) and game performance compared to the non-debugging version.

2. Participants in the Pirate Plunder groups would improve their abstraction and decomposition scores (measured by Dr. Scratch) from pre-to post-test compared to the Scratch control group.

8.4 Results

8.4.1 Hypothesis 1 - Did the Debugging-First Approach Have a Positive Impact on Learning Outcomes and Pirate Plunder Performance?

Hypothesis 1 was that the debugging-first version of Pirate Plunder would have a positive impact on learning outcomes (Dr. Scratch abstraction on the Scratch task assessment and CTt scores) and game performance compared to the non-debugging version.

Scratch Task - Abstraction

Figure 8.8 shows the mean Dr. Scratch abstraction and decomposition learning gains from pre-to post-test for each group (Table 8.4 shows the descriptive statistics). A one-way ANOVA showed no significant difference between the learning gains of the three groups, $F(2, 67) = 1.01$, $p = .340$, $\eta^2 = .032$.

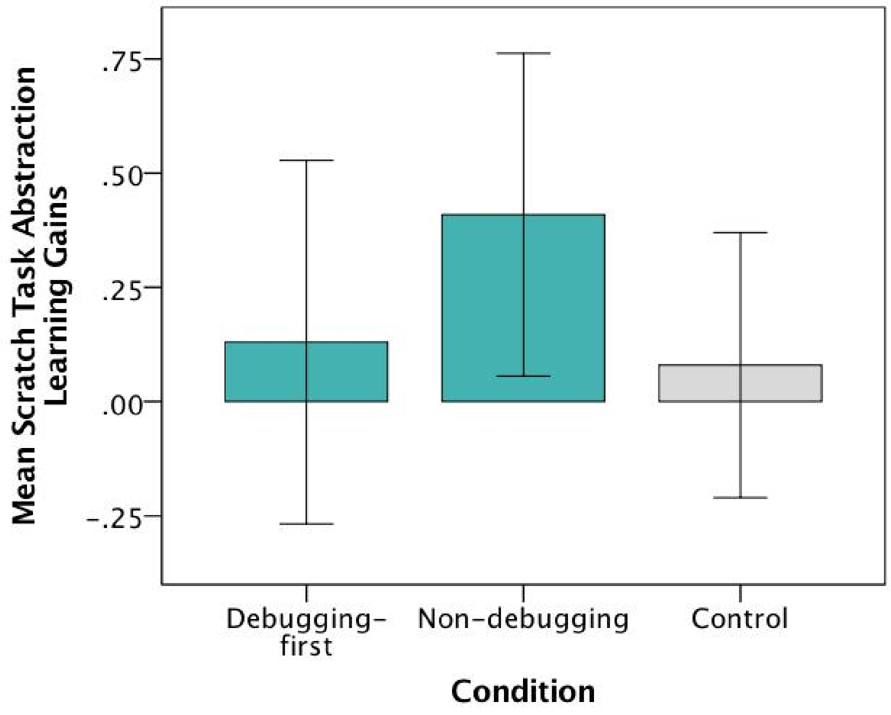


Figure 8.8: Comparison of the Dr. Scratch abstraction and decomposition learning gains on the Scratch task assessment from pre-to post-test for each group (error bars show 95% confidence interval)

In the Scratch task post-test, seven of the 70 participants attempted to use custom blocks and/or cloning (10%), with six of these in the intervention condition (Table 8.5). The two intervention groups were identical in terms of the number of participants that attempted to use custom blocks and cloning. Dr. Scratch cannot assess whether

Table 8.4: Descriptive statistics of the Dr. Scratch abstraction and decomposition scores on the Scratch task assessment (maximum score of 3)

Condition			Pre-Test	Post-Test	Learning Gains
Pirate Plunder	Debugging-first	*M*	0.74	0.87	0.13
		N	23	23	23
		SD	0.54	0.69	0.92
	Non-debugging	*M*	0.82	1.23	0.41
		N	22	22	22
		SD	0.40	0.69	0.80
Control		*M*	1.00	1.08	0.08
		N	25	25	25
		SD	0.58	0.40	0.70

this functionality was used correctly, but this can be measured using the expected outcome score (Section 8.4.4). Of the six participants in the intervention condition who used custom blocks and/or cloning, four achieved a full expected outcome score of 4 (one debugging-first and three non-debugging) and two achieved 0 (one from each group).

Table 8.5: Breakdown of the post-test Dr. Scratch abstraction and decomposition scores for each group

Condition	*N*	Dr. Scratch Abstraction and Decomposition Score			
		0	1	2	3
Debugging-first	23	6	15	1	1
Non-debugging	22	1	17	2	2
Control	25	0	24	0	1
Total	70	7	56	3	4

Computational Thinking Test

Figure 8.9 shows the mean Computational Thinking test learning gains from pre- to post-test for each group (Table 8.6 shows the descriptive statistics). A one-way ANOVA showed no significant difference between the learning gains of the three groups $F(2, 72) = 0.28$, $p = .76$, $\eta^2 = .008$.

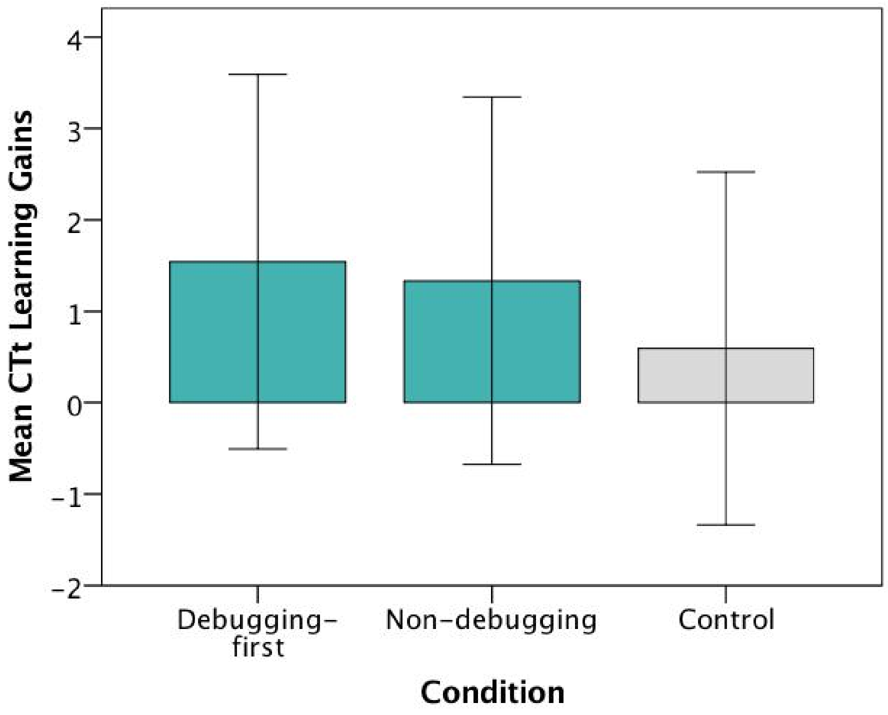


Figure 8.9: Comparison of the Computational Thinking test learning gains from pre-to post-test for each group (error bars show 95% confidence interval)

Table 8.6: Descriptive statistics of the Computational Thinking test for each group (maximum score of 28)

Condition			Pre-Test	Post-Test	Learning Gains
Pirate Plunder	Debugging-first	*M*	15.42	16.96	1.54
		N	24	24	24
		SD	4.05	4.80	4.85
	Non-debugging	*M*	16.63	17.96	1.33
		N	24	24	24
		SD	4.27	5.00	4.76
Control		*M*	17.96	18.56	0.59
		N	27	27	27
		SD	4.41	4.39	4.88

Pirate Plunder Performance

Pirate Plunder performance is judged by the number of challenges completed (out of 40) and overall stars collected (a maximum of 120, 3 per challenge). The average stars collected is given as an additional measure of performance (out of 3) but is not part of the main measurement. Both groups spent 290 minutes playing the game (4 hours 50 minutes).

Table 8.7 shows a breakdown of these statistics for both Pirate Plunder groups. Because there are two measures of performance, the required confidence level for significance is increased to 97.5% (p < 0.025). In this case, there was no difference between-groups in the number of challenges completed, $t(55)$ = 1.96, p = .055, d = 0.51, or the total stars collected, $t(55)$ = -2.01, p = .049, d = 0.53.

Table 8.7: Descriptive statistics of Pirate Plunder performance for both intervention groups

Condition		Challenges Completed	Total Stars Collected	Average Stars per Level
	M	31.28	91.62	2.91
Debugging-first	*N*	29	29	29
	SD	5.30	17.87	0.16
	M	34.04	100.79	2.96
Non-debugging	*N*	28	28	28
	SD	5.35	16.42	0.08

8.4.2 Hypothesis 2 - Did Pirate Plunder Improve Abstraction in Scratch Compared to the Control?

Hypothesis 2 was that the Pirate Plunder groups would improve their abstraction scores (measured by Dr. Scratch) from pre-to post-test more so than the Scratch control group. Using an independent samples t-test, there was no significant difference in the learning gains between the intervention and the control, $t(68)$ = 0.92, p = .36, d = 0.24 (Table 8.8).

Table 8.8: Descriptive statistics of the Dr. Scratch abstraction and decomposition scores on the Scratch task for the intervention and control groups (maximum score of 3)

Condition		Pre-Test	Post-Test	Learning Gains
	M	0.78	1.04	0.27
Intervention	*N*	45	45	45
	SD	0.47	0.71	0.86
	M	1.00	1.08	0.08
Control	*N*	25	25	25
	SD	0.58	0.40	0.70

As previously mentioned in Section 8.4.1, only seven of the 70 participants (10%) who did the Scratch task at post-test attempted to use custom blocks and/or cloning, with six of these in the intervention condition. Most participants completed the task using ‘show’, ‘hide’ and ‘wait’ blocks to get each object to appear, using four object sprites in the process.

8.4.3 Artifact-Based Interviews Observations

Participants were selected for the interviews based on their use of custom blocks and cloning on the Scratch task assessment in relation to their Pirate Plunder progress, or for solutions with obvious duplication. During the interviews (*N* = 16), it became apparent that when prompted, participants could explain what custom blocks are, why they are used and where they could have used them in the Scratch task. For example:

> **Researcher:** "Could you have used a custom block?"
> **Participant:** "Oh yeah – I might have been able to actually – instead of doing a repeat and move every time, I could've had a custom block to move and it would move as many steps as it needed to move."
> **Researcher:** "Would you have put the turn in it?"
> **Participant:** "In the cat, yes because it has to turn every time."
> **Researcher:** "What is the advantage of using a custom block?"
> **Participant:** "It makes it easier because you don't have to keep clicking repeat and move. And if you add inputs it can move as many as you want it to."

Several of these participants could also explain what cloning is and why it is used in Scratch, but had difficulty applying this knowledge to the Scratch task assessment. Various reasons were given for not using custom blocks and cloning in the post-test, including the lack of a block limit, Pirate Plunder working differently to Scratch and wanting to complete the task using a similar method to the method they had used at pre-test. These reasons are explored in the discussion (Section 8.5).

8.4.4 Differences in Scratch Task Performance

An expected outcome score out of 4 (Table 8.3) was used to establish whether participants had produced a project that met the specification of the Scratch task. There was no significant difference in the expected outcome learning gains between groups, $F(1, 68) = 0.05$, $p = .82$, $\eta^2 = .001$. Table 8.9 shows the descriptive statistics. As the pre-test scores of the three groups were significantly different ($F(2, 67) = 7.87$, $p = .001$, $\eta^2 = .19$), paired samples t-tests show significant improvements for the debugging-first group ($t(22) = 2.69$, $p = .013$, $d = 0.68$), the non-debugging group ($t(21) = 5.08$, $p < .001$, $d = 1.06$) and the control group ($t(24) = 4.54$, $p < .001$, $d = 1.01$).

Table 8.9: Descriptive statistics of the expected outcome scores on the Scratch task for each group

Condition			Pre-Test	Post-Test	Learning Gains
Pirate Plunder	Debugging-first	*M*	0.96	1.96	1.00
		N	23	23	23
		SD	1.15	1.71	1.78
	Non-debugging	*M*	1.27	2.86	1.59
		N	22	22	22
		SD	1.39	1.58	1.47
Control		*M*	2.32	3.52	1.20
		N	25	25	25
		SD	1.22	1.16	1.32

8.4.5 Differences in Post-Test Confidence

Table 8.10 shows the descriptive statistics for the Pirate Plunder and Scratch post-test questionnaire confidence ratings. The confidence ratings are on a scale from 3 to 0 (very confident to not confident) and confidence changes from 1 to -1 (improved to declined) (Section 8.3.3).

There were significant differences in Scratch confidence ($F(2, 67) = 3.95$, $p = .024$, $\eta^2 = .11$), Scratch confidence change ($F(2, 67) = 9.81$, $p < .001$, $\eta^2 = .23$), custom block confidence ($F(2, 67) = 6.41$, $p = .003$, $\eta^2 = .16$), custom block confidence change ($F(2, 67) = 3.62$, $p = .032$, $\eta^2 = 0.1$) and clone confidence change ($F(2, 67) = 5.5$, $p = .006$, $\eta^2 = 0.14$). All of these differences favoured the non-debugging group or the control and the non-debugging group over the debugging-first group.

Table 8.10: Descriptive statistics of the questionnaire confidence ratings for each group (PP = Pirate Plunder, DF = Debugging-first and ND = Non-debugging)

Condition		Scratch Confi-dence	Change	Custom Block Confi-dence	Change	Clone Confi-dence	Change
PP DF	*M*	1.54	0.46	1.21	0.38	1.00	0.25
	N	24	24	24	24	24	24
	SD	0.88	0.72	1.10	0.71	0.78	0.79
PP ND	*M*	1.96	1.00	2.08	0.81	1.46	0.81
	N	26	26	26	26	26	26
	SD	0.72	0.00	0.94	0.49	0.86	0.40
Control	*M*	2.20	0.90	1.20	0.45	1.55	0.40
	N	20	20	20	20	20	20
	SD	0.77	0.31	0.95	0.61	0.83	0.60

8.4.6 Differences in Pirate Plunder Analytics

Section 8.4.1 showed that there were no differences in Pirate Plunder performance between groups. This section analyses the Pirate Plunder analytics in more detail, examining level attempts, time spent on each level and program manipulation.

Table 8.11 shows the average attempts and time spent statistics, both overall and per level, for each version. The debugging-first group had significantly more total level attempts ($t(55) = 2.41$, $p = 0.019$, $d = 0.55$) and average attempts per level ($t(55) = 3.51$, $p = .001$, $d = 0.93$). Yet, the average time per level was not significantly different. Level attempts with a time under one second were removed from the dataset because these indicate that the button was spammed and would skew results.

Table 8.11: Descriptive statistics of Pirate Plunder attempts and time for each intervention group

Condition		Total Level Attempts	Average Attempts per Level	Total Level Time (HH:MM:SS)	Average Time per Level (Seconds)
	M	276.97	8.93	02:58:25	343.04
Debugging-first	*N*	29	29	29	29
	SD	116.03	3.45	00:40:49	63.43
	M	211.29	6.18	03:05:30	325.8
Non-debugging	*N*	28	28	28	28
	SD	86.93	2.36	00:47:13	75.11

Table 8.12 shows the program manipulation statistics (block additions, deletions and moves) for each group. Interestingly, the average manipulation per attempt was significantly higher for the non-debugging group ($t(55) = 2.05$, $p = .045$, $d = 0.54$), whereas average manipulation per level was significantly higher for the debugging-first group ($t(55) = 2.41$, $p = .02$, $d = 0.64$).

Table 8.12: Descriptive statistics of Pirate Plunder program manipulation for each intervention group

Condition		Total Program Manipulation	Average Program Manipulation per Attempt	Average Program Manipulation per Level
	M	3256.80	12.31	102.88
Debugging-first	*N*	29	29	29
	SD	1409.35	3.97	37.15
	M	2890.67	14.71	83.46
Non-debugging	*N*	28	28	28
	SD	887.84	4.83	21.45

8.4.7 CT as a Predictor of Pirate Plunder Performance

The CTt pre-test scores correlated significantly with both measures of Pirate Plunder performance: number of challenges completed, *r*(53) = .44, *p* = .001 (Figure 8.10) and total stars collected, *r*(53) = .42, *p* = .002 (Figure 8.11).

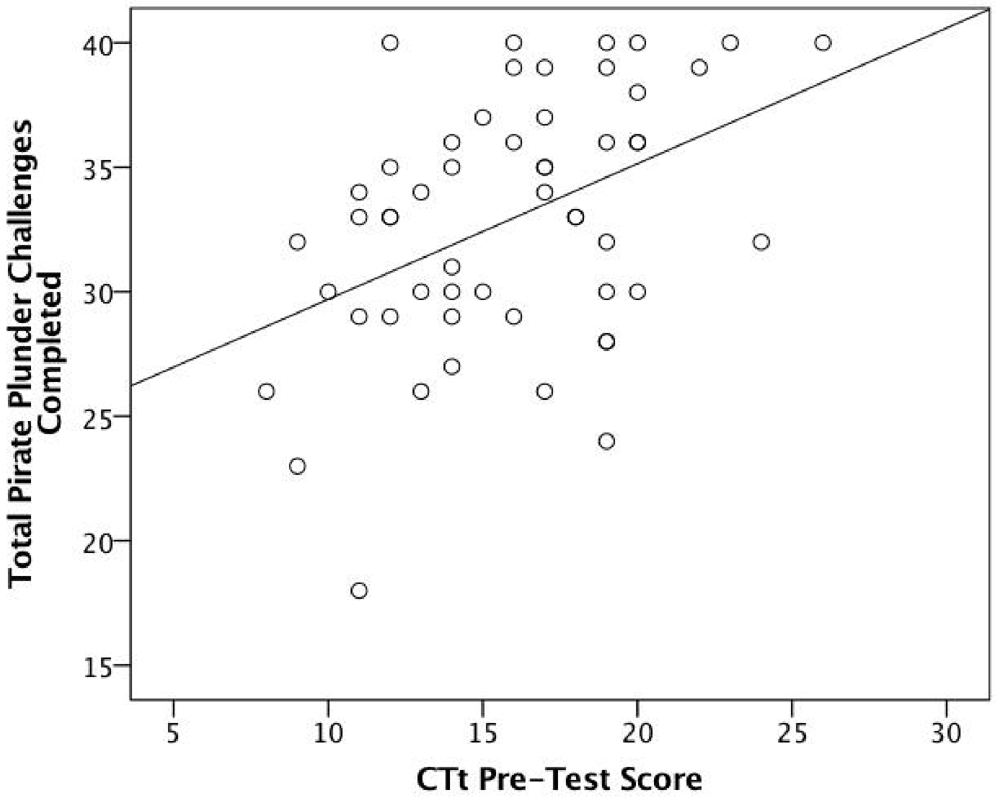


Figure 8.10: Relationship between CTt pre-test score and number of Pirate Plunder challenges completed for each participant (with regression line)

8.4.8 Scratch Expected Outcome Learning Gains as a Predictor of CT Learning Gains

The CTt was included as a measure to see if improvements in Scratch programming would correspond to improvements in CT. The Scratch task expected outcome measure is a good indicator of Scratch programming ability as it assesses whether the participant can produce a project to a solution. There was no correlation between the learning gains of the two measures, *r*(69) = .15, *p* = .228. However, participants did improve on the CTt overall after both programming interventions (Section 8.4.1).

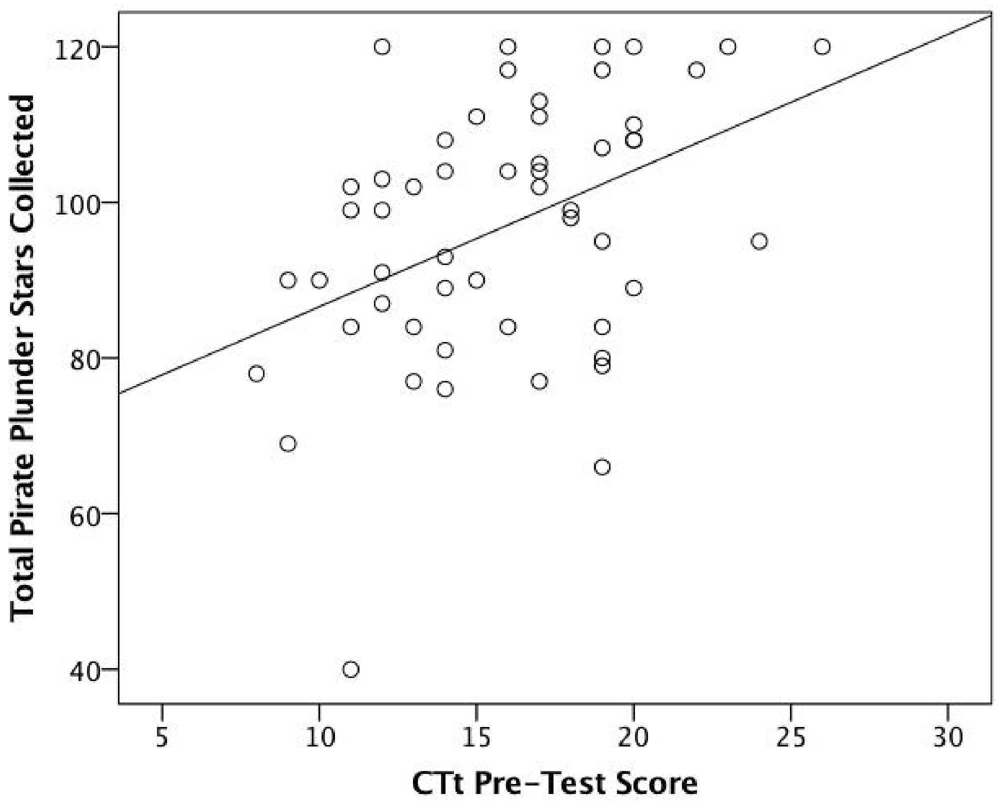


Figure 8.11: Relationship between CTt pre-test score and number of Pirate Plunder stars collected for each participant (with regression line)

8.5 Discussion

8.5.1 Was Debugging-First Beneficial?

The results show that a debugging-first approach was not beneficial in Pirate Plunder in comparison with a non-debugging approach, in terms of learning gains on the Scratch task assessment, CTt and the Pirate Plunder performance statistics. This suggests that the debugging-first blocks were no more beneficial than starting with an empty program on each level.

Debugging-First Constraints

The debugging-first blocks were designed to both fit the difficulty progression (aiding the player) and to address the antipatterns observed by Lee et al. (2014). However, during the study, it was observed that players would be confused by the debugging-first blocks and often attempt to clear the program, even if some blocks were undeletable. The results also show that on average, the players in the debugging-first condition manipulated their programs more and had more total attempts and attempts per level. This was expected, as the player has to come to grips with the debugging-first program. Yet, this did not have the expected positive impact on the learning outcomes. This suggests that the style of debugging-first blocks used may have been too restrictive and actually hindered the player instead of helping them.

During the artifact-based interviews, participants in the debugging-first group were asked about how helpful they found these blocks. Several participants responded that they were only useful sometimes and would often get in the way. For example:

> **Researcher:** "In Pirate Plunder, you had these locked blocks, did you think they were helpful? Or unhelpful?"
> **Participant:** "So sometimes I moved them around and didn't use them."
> **Researcher:** "Did you think they were helpful?"
> **Participant:** "Yeah, they were helpful because they were giving you a vague idea of what you had to do. I just played it and tried to figure out what they were doing."
> **Researcher:** "Would you have preferred not having them at all?'
> **Participant:** "Sometimes – sometimes not. Cos on some of them I found them harder than others and they all had those on. Like sometimes it would've been easier with those on and sometimes not."

8.5.2 Did Pirate Plunder Work?

The results show that there was no difference between the Dr. Scratch abstraction and decomposition scores for the intervention and control groups on the Scratch task assessment. However, when questioned during the artifact-based interviews, intervention participants could explain how they would use custom blocks, and in some cases cloning, on the task. When asked why they did not use these, participants said that they did not know that they had to. This indicates that the Scratch task was not a good enough assessment to show participant understanding of the learning content. The possible reasons for this are discussed below.

No Motivation for Using Abstraction in the Scratch Task

The first reason for the lack of abstraction on the Scratch task assessment is that the task does not motivate participants to use custom blocks and cloning. Whereas in Pirate Plunder, they have to use abstraction to complete levels and this is enforced by motivational strategies such as a block limit and required blocks (Section 7.4.2). For example:

> **Researcher:** "Could you have used a custom block in your solution?"
> **Participant:** "I'm not quite sure – because Scratch is a bit different to Pirate Plunder. You probably could because there is options for custom blocks – but I don't really know how you would use them, because you don't really have the limited blocks. Because there's no block limit then you wouldn't really need too – but you could just to make it a bit quicker."

In comparison, the objective of the Scratch task is to achieve the required behaviour of moving the cat sprite around the rectangle. This does not have to be done using the same method as Pirate Plunder and can be done using other 'Move' category blocks such as 'glide to position'. In addition, the project will only contain duplication, and motivation for abstraction, if the participant produces it. This means that the participant has to first complete the task and then see that they can refactor their solution, which is a lot of functionality to produce in the 40 minutes they were given for the task.

Same Scratch Task at Pre-and Post-test

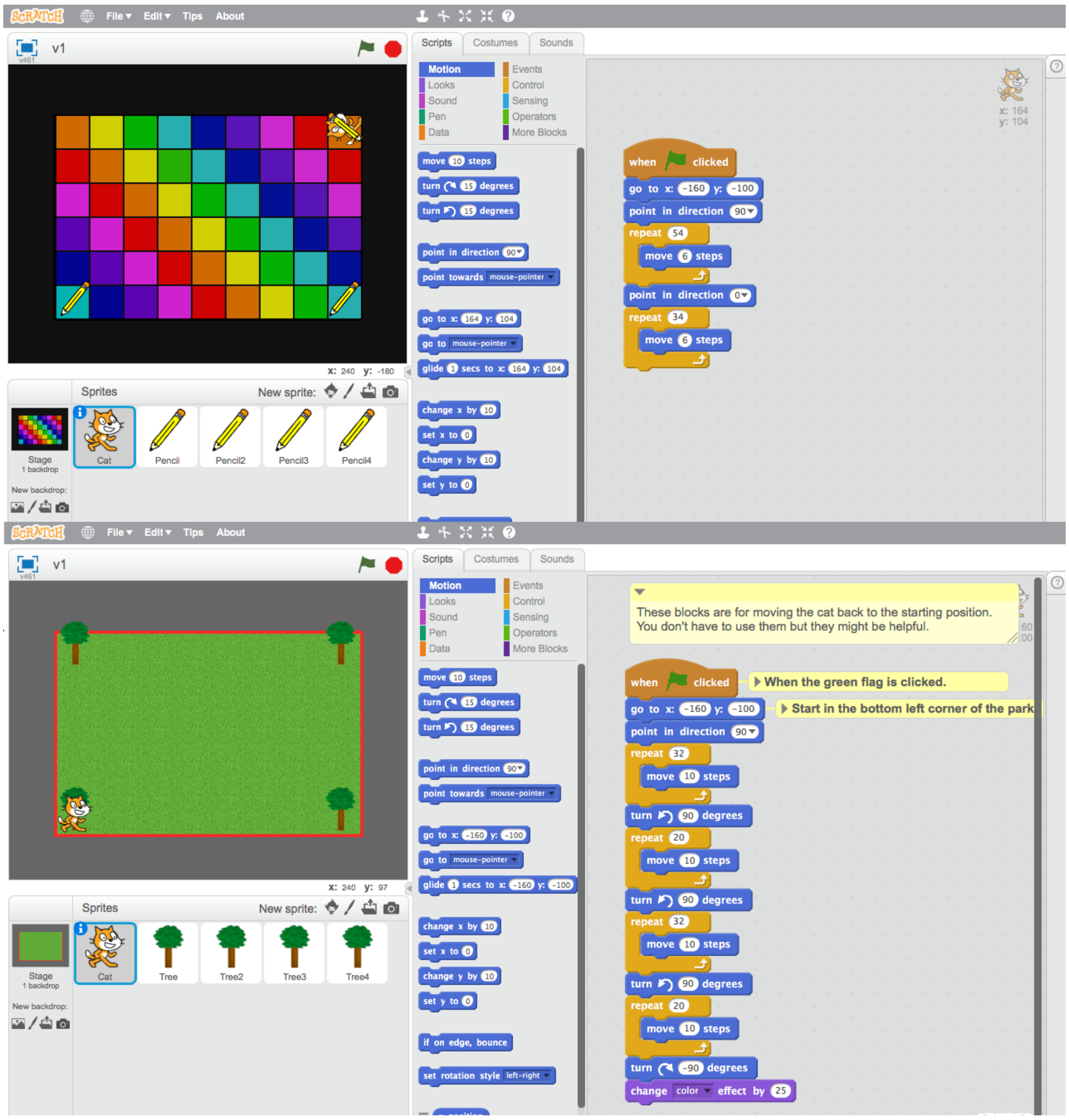
The second possible reason is that because the participants were doing a similar assessment at pre-and post-test, they completed the task at post-test using a similar method as they did at pre-test. This is supported by research into memory strategies, which suggests that children will stick to an existing strategy instead of using a new one because it requires less cognitive effort (Lodico, Ghatala, Levin, Pressley, & Bell, 1983). Participants would often use the functionality that they had attempted to use at pre-test instead of attempting to use the new functionality. An example of this is shown in Figure 8.12, where a participant (from the non-debugging group) created an incomplete solution at pre-test that used repeat blocks. At post-test, they then finished this solution using the same, but now complete, repeat blocks. It is worth noting that this participant had reached level 33 of Pirate Plunder, completing all the custom block and inputs levels, so in theory, should have been able to use them in Scratch.

Scratch Differences to Pirate Plunder

During the interviews, some participants stated that Scratch was too different from Pirate Plunder and that this affected their solution to the Scratch task assessment at post-test. Participants gave two main reasons for this, which are described below.

The first reason was that there is no resetting in Scratch. Whereas in Pirate Plunder, the grid resets after each program run, Scratch sprites will maintain their state from the previous program run. Despite the starting project (Figure 8.5) having blocks to reset the cat sprite and comments to explain this, many participants simply deleted these blocks and attempted to start again. They then got confused when the cat sprite would not return to the bottom left corner of the park/room once their program had run.

The second reason was the grid differences, namely the much larger Cartesian system in Scratch (Section 7.4.3) and the coordinates on the x-and y-axes in Pirate Plunder. This meant that distances were considerably bigger in Scratch and participants had to obtain grid coordinates by using the coordinate position indicator underneath the stage that updates as the player moves their mouse around the grid. Despite Pirate Plunder having this, players rarely used it because of the x-and y-coordinates on the grid axes.



```
when [flag] clicked
go to x: -160 y: -100
point in direction 90
repeat 54
    move 6 steps
point in direction 0
repeat 34
    move 6 steps
```

These blocks are for moving the cat back to the starting position. You don't have to use them but they might be helpful.

```
when [flag] clicked                    When the green flag is clicked.
go to x: -160 y: -100                  Start in the bottom left corner of the park
point in direction 90
repeat 32
    move 10 steps
turn ↺ 90 degrees
repeat 20
    move 10 steps
turn ↺ 90 degrees
repeat 32
    move 10 steps
turn ↺ 90 degrees
repeat 20
    move 10 steps
turn ↻ -90 degrees
change color effect by 25
```

Figure 8.12: Screenshot of a participant Scratch project attempting to complete the task using the same solution at pre-and post-test, not considering attempting to use custom blocks or cloning

Not Remembering the ‘Property Of’ Block

When questioned on how they would use cloning blocks, participants struggled to describe how they would get the object to appear in the same position as the cat sprite. For example:

> **Researcher:** “Could you have used cloning in this project?”
> **Participant:** “I think so – to maybe clone the trees.”
> **Researcher:** “What would the advantage of that be?”
> **Participant:** “You wouldn’t have to use the buttons as well and it would just do it itself.”

> **Researcher:** “How would you go about getting the clone to go to the right position? Would you still need four trees?”
> **Participant:** “No – when you clone something it means there’s more than one of the thing. I’d use the go to position block to get it to go to the right position.”
> **Researcher:** “How would you get each tree clone to go to the right place, if you can only give it one set of numbers?”
> **Participant:** “You could put in the go to block – I can’t remember. Would you use delete clone?”

Potentially, this is because the ‘property of’ block in the ‘Sensing’ category is introduced as part of the ‘cloning other sprites’ tutorial, where the player is already taking on a lot of new information.

8.5.3 Limitations

Other than the issues with Pirate Plunder and the Scratch task assessment described in the previous section, another limitation of the study was that the non-debugging group were better than the debugging-first group at the start of the study, as shown by their higher pre-test scores on both the Scratch task assessment and the CTt. Yet, this difference was difficult to avoid because the school required the use of class groups who may have had different teaching experiences before the study.

8.5.4 Pirate Plunder Feedback

Participants were asked to give feedback on Pirate Plunder as part of the questionnaire (Appendix M) and at the end of the artifact-based interviews. The majority of suggestions for improving the game involved getting better help from the feedback parrot. There were also some suggestions for removing the five-coin update cost in the shop and adding extra shop items for female pirates (dresses, haircuts, etc.)

8.5.5 Non-Programming Control

To confirm that Pirate Plunder is effective in meeting its learning outcomes, a non-programming control group would be useful in providing a comparison to a group that did no programming (and therefore no CT) during the study. This would test whether Pirate Plunder works in terms of improving abstraction scores and expected outcomes, compared to a group that did no block-based programming. This second control group would still be active, and would still be doing a technology-based task, to ensure that they are a reliable comparison to the intervention group. This was done in Study 4 (Chapter 9).

8.6 Conclusions

In conclusion, the debugging-first approach in Pirate Plunder was not beneficial to players. The debugging-first blocks may have been too restrictive, particularly the undeletable and unchangeable inputs. Not allowing the player to clear the program, in an attempt to address the 'reinventing the wheel' antipattern, may have been detrimental to some players. Consequently, Study 4 will use a single version of Pirate Plunder that contains some debugging-first blocks (those that the game analytics showed were useful) that can be deleted if the player does not want to use them.

There was also no difference between the experimental condition (Pirate Plunder) and the active control (Scratch curriculum) in Dr. Scratch abstraction scores on the Scratch task assessment from pre-to post-test. This could be down to weaknesses in the task. The artifact-based interviews suggested that participants had understood how to reuse Scratch code using custom blocks and cloning, but the Scratch task assessment was not good enough to pick up these differences. The lack of a difference between the expected outcome scores on the assessment suggests that Pirate Plunder is effective in teaching children to use basic Scratch functionality, or at least enough to complete a simple Scratch task.

The weaknesses of the Scratch task assessment discussed in Section 8.5.2 include not motivating the player to use abstraction and the task being the same at pre- and post-test. Weaknesses with Pirate Plunder include differences to Scratch and participants not remembering how to use the 'property of' block. Several changes are made to the Scratch task assessment in Study 4 to address these weaknesses, including moving the blocks to reset the position of the cat sprite to a different starting event. Study 4 used a new post-test assessment that explicitly asks the player to reduce the number of blocks in a pre-made project. This is discussed in more detail in the next chapter (Chapter 9).

The next chapter describes a study designed to evaluate the overall efficacy of Pirate Plunder to teach abstraction skills to children in Scratch. It builds on the results of this study, using a crossover design, a non-programming control group, improved assessments and an updated version of the game. The updated version of Pirate Plunder has an improved help function, name customisation (so that player names are not stored in the database), a revised difficulty progression (with looser debugging-first blocks) and more shop items.

Chapter 9

Study 4 - Using Pirate Plunder to Improve Children's Abstraction Skills in Scratch

This chapter describes a study designed to evaluate whether Pirate Plunder can be used to teach primary school children age 10 and 11 how to use abstraction in Scratch. The observations, results and participant feedback from Study 3 (Chapter 8) have been used to develop an updated version of Pirate Plunder, produce revised assessments and devise a new experimental design.

The chapter begins by describing the conclusions and changes made from Study 3. It then describes a pilot study before giving the results, discussion and conclusions of the study reported in this chapter.

9.1 Introduction

This section explains how the conclusions from Study 3 have been used to create an updated version of Pirate Plunder. It then describes how the observations and results have been used to create two new Scratch abstraction assessments and an updated experimental design, which are described in detail in Section 9.3.3.

9.1.1 Conclusions from Study 3

Study 3 was designed to investigate the value of a debugging-first approach in Pirate Plunder and to measure whether Pirate Plunder was effective in teaching children to use abstraction skills in Scratch.

Debugging-First

Study 3 was unable to demonstrate any benefits of the debugging-first version over the non-debugging version of Pirate Plunder. Observations during the intervention and responses during the artifact-based interviews indicated that participants found them too restrictive and got frustrated when they did not understand them. Particularly the undeletable and unchangeable input blocks that were added to deal with the 'reinventing the wheel' antipattern identified by Lee et al. (2014).

Abstraction

There was no difference observed between the Pirate Plunder groups and the Scratch active control group on the tasks designed to measure abstraction in Scratch. Yet, when the same participants who had played Pirate Plunder were interviewed about using custom blocks, most could explain what they were and how they could have used them in the task. This suggests that there were weaknesses in the Scratch assessment, specifically a lack of motivation for using abstraction and participants producing similar solutions at pre-and post-test because the task was the same.

Pirate Plunder was successful in getting players to use custom blocks and cloning within the game. This indicates that the game mechanics explained in Chapter 7 work and can be built upon for the study reported in this chapter. Yet, there were issues with participants being unable to recall the 'get property of' block and complaints about the hint function.

Computational Thinking

Study 3 used the CTt (Computational Thinking test) as an additional measure to see whether improvements in Scratch programming corresponded to improvements in CT. There were overall improvements from pre-to post-test, but no difference between groups. This was expected because both groups were applying computational thinking (CT) through programming, in line with the current literature.

9.1.2 Pirate Plunder Changes

Several changes were made to Pirate Plunder before the start of Study 4. These were:

1. Removing the restrictive debugging-first blocks and leaving in looser debugging-first blocks that were useful to players in Study 3.
2. Updating the difficulty progression.
3. Adding a tutorial for the 'get property of' block.

4. Adding an improved help function.
5. User interface changes to make Pirate Plunder more similar to Scratch.
6. Name customisation to remove players' real names for GDPR.

All these changes were combined into a single version of Pirate Plunder that was used in the study reported in this chapter. This version was tested in a pilot study before Study 4 began (Section 9.2).

Debugging-First Blocks

Debugging-first blocks were altered considerably for Study 4 due to the lack of benefit for players using them in Study 3. This was done using player feedback, observations and level analytics. The blocks were altered in two ways: removing restrictive block types and only having debugging-first blocks on levels where they were beneficial in Study 3.

The four types of debugging-first blocks from Study 3 were as follows (Section 8.1.2):

1. Blocks with an incorrect input.
2. Blocks with a locked (but correct) input.
3. Assistance blocks that are undeletable.
4. Assistance blocks that not locked but are sometimes not needed in the solution.

For Study 4, undeletable assistance blocks (Type 3) were removed completely and blocks with a locked (but correct) input (Type 2) were unlocked so that the player could change their value. In the Study 3 post-test interviews, participants stated that the debugging-first blocks could be confusing and locking them made some levels more difficult. This was supported by the higher number of total level attempts and average attempts per level for the debugging-first players (Section 8.4.6). Combined with their lower performance overall, this indicates that they found levels more difficult because of the restrictive debugging-first blocks. Block Types 1 and 4 were left in the game, leaving debugging-first code that is 'looser' and can be removed by the player (Figure 9.1 shows an example). The debugging-first blocks for a challenge level can be brought back using the 'reset' button.

This may now encourage the 'reinventing the wheel' antipattern, where players delete the debugging-first code without reading or understanding it (the rationale for the undeletable blocks). However, this decision was made as a trade-off between the frustration that players feel because they do not understand the debugging-first functionality, instead being guided to the solution themselves using tutorials, previous level solutions and the new help feature.

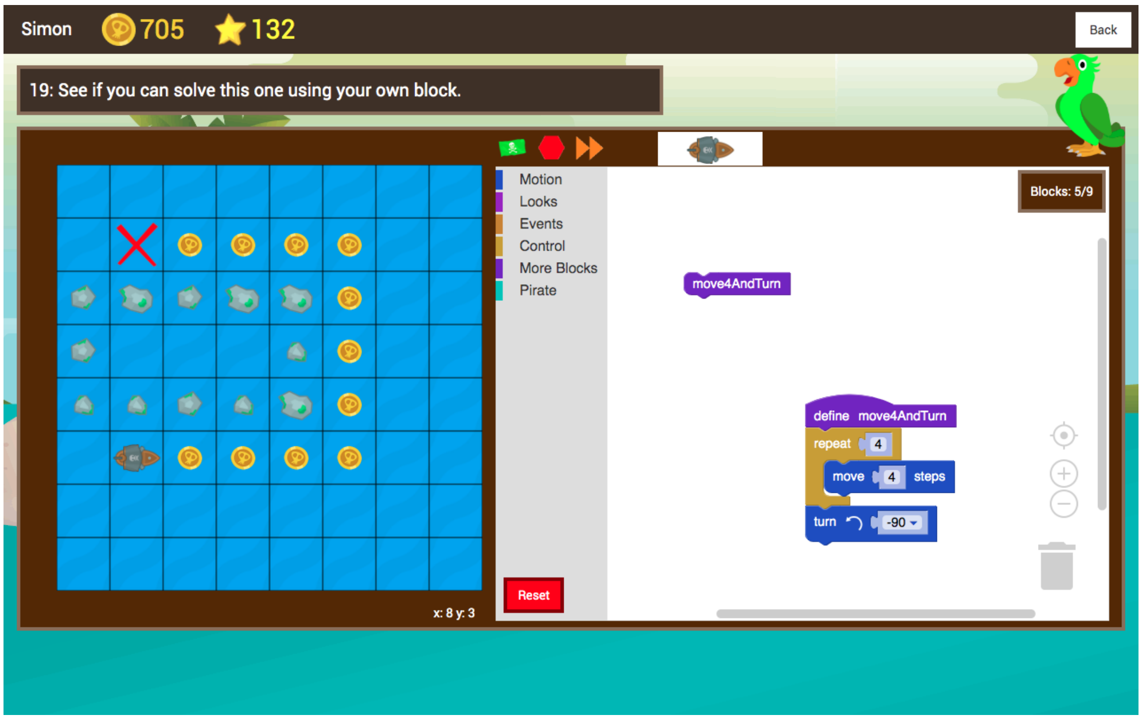


Figure 9.1: Screenshot of an example of the updated debugging-first instructions for Study 4, all the blocks can be removed, and the inputs can be changed

The decision was made to leave the 'looser' debugging-first blocks on the levels in which they were beneficial to players in Study 3. This was judged on a level-by-level basis using the following criteria. On average, did the debugging-first players:

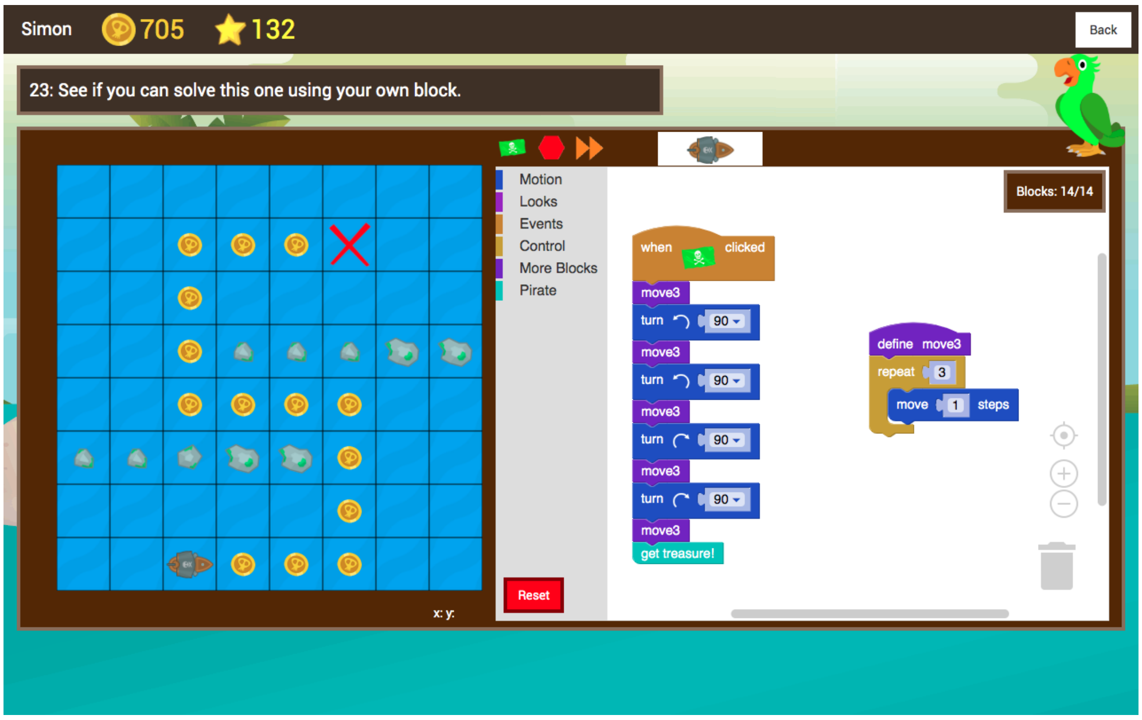
- Collect more stars?
- Collect more coins?
- Take fewer attempts to complete the level successfully?
- Take less time to complete the level successfully?

If the level met three or more of these requirements, the debugging-first blocks were left in with restrictive block types removed. Blocks were also retained for the first one or two challenges per tutorial, to give the player further examples of how to use the tutorial block. The resulting level progression is shown in Appendix P.

Updated Difficulty Progression

Game analytics and observations from Study 3 were used to make a minor alteration to the difficulty progression. It was observed that players often got stuck on level 22 and had to ask for help. The game analytics showed that this was one of the levels with the overall lowest average star count (2.75/3) and the lowest percentage of collected coins (86.23%), other than the first 8 levels where the player is learning the fundamentals of the game.

The difficulties with level 22 arise because it requires the player to recognise that 'move3' is the required custom block because the turns are in different directions (as shown in Figure 9.2). Often, players would create a 'move3AndTurn' block that moves the ship and turns in a single direction, similar to that used in levels 20 and 21. It was decided that swapping levels 22 and 23 would give the player more practice in using custom blocks and recognising when turns are required inside or outside of them.



```
when [flag] clicked
move3
turn ↺ 90
move3
turn ↺ 90
move3
turn ↻ 90
move3
turn ↻ 90
move3
get treasure!

define move3
repeat 3
    move 1 steps
```

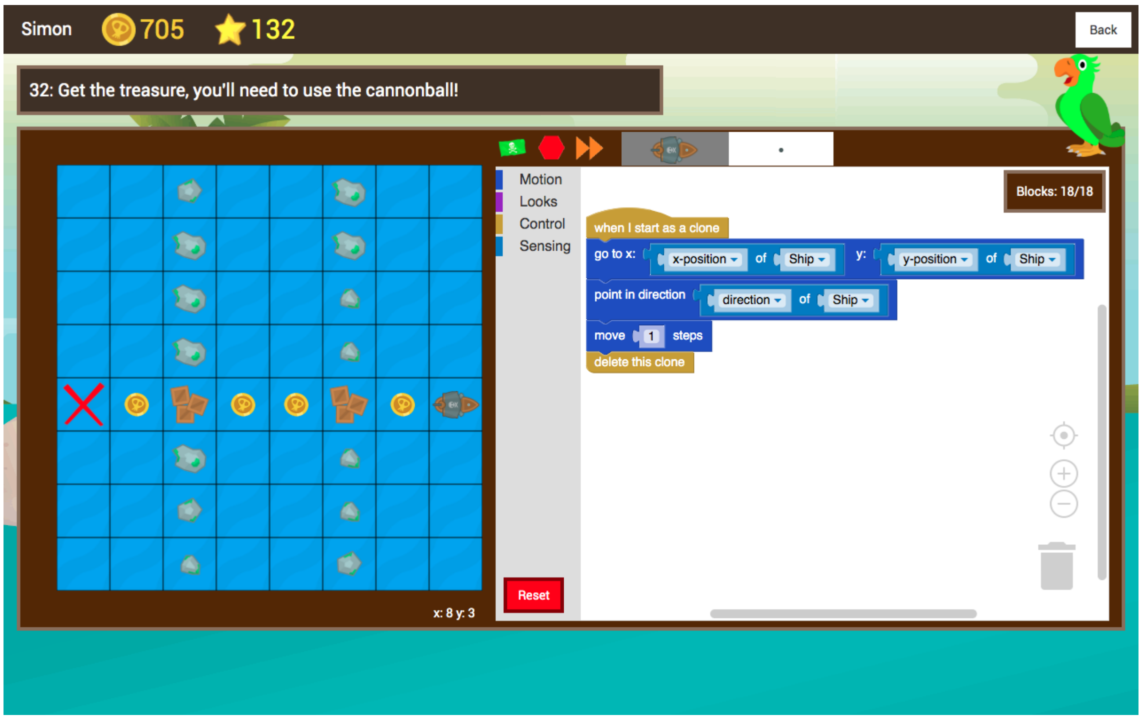
Figure 9.2: Screenshot of the ideal solution to challenge 23 (previously challenge 22) in the Study 4 version of Pirate Plunder

'Get Property Of' Tutorial

The post-test Scratch task assessment projects and artifact-based interviews from Study 3 showed that participants had difficulty recalling how to clone objects in the current location of a sprite. This is done using the 'get property of' block, which can be used to get the current coordinate position of a sprite in Scratch. In Pirate Plunder, the player must use the block to move the cloned cannonball sprite to the position of the ship sprite and then use it again to set it to the direction of the ship, ensuring that it is fired (or moved) in the same direction as the ship (Figure 9.3).

'Get property of' was introduced in the Study 3 version of Pirate Plunder as part of the 'cloning other sprites' tutorial. The fact that it was not used by participants in the post-test Scratch task, and that some interview participants struggled to recall it, suggested that the block needed a separate tutorial.

The 'get property of' tutorial was added in between the 'cloning myself' and 'cloning other sprite' tutorials. In terms of the level progression, it acts as an extra cloning tutorial and does not unlock any challenges on its own. In the tutorial, the player is


Simon 705 132 Back

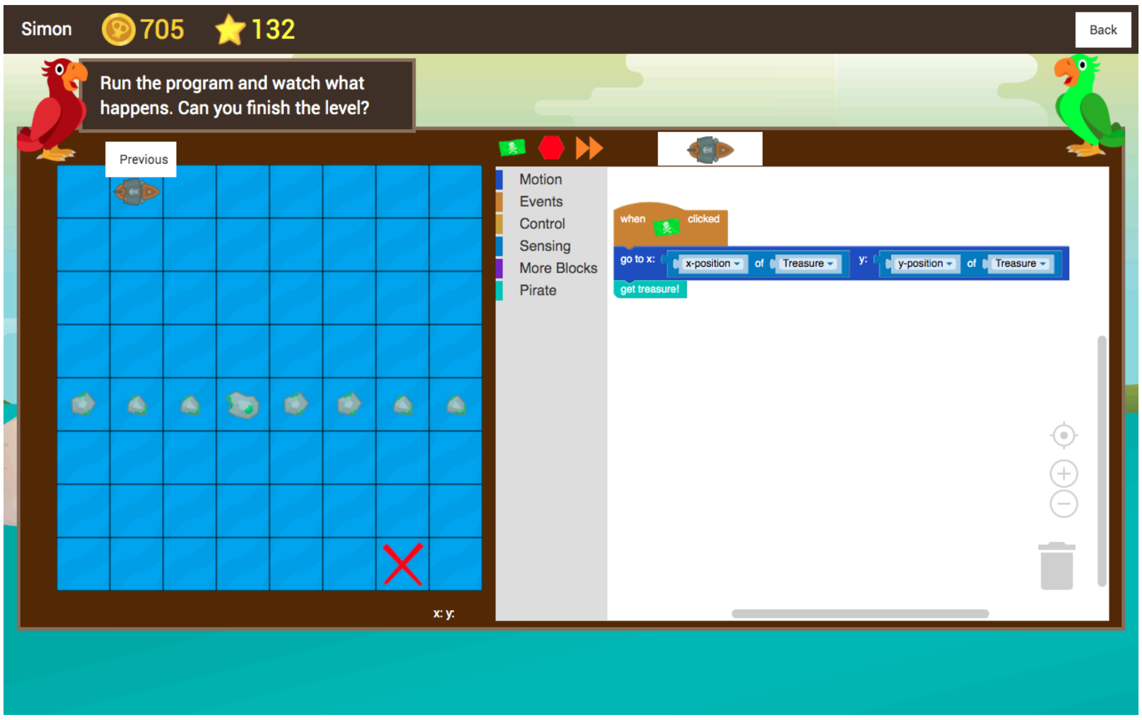
32: Get the treasure, you'll need to use the cannonball!

Blocks: 18/18

```
when I start as a clone
go to x: (x-position of Ship) y: (y-position of Ship)
point in direction (direction of Ship)
move 1 steps
delete this clone
```

Figure 9.3: Screenshot of the ‘get property of’ block used in the cannonball sprite in Pirate Plunder

introduced to the ‘sensing’ block category. They must navigate the ship to avoid the rocks in the middle of the map (Figure 9.4). This can only be done using the ‘go to position’ block to teleport the ship to the treasure (because the ship will crash if the ‘move’ block is used to go over them.) The ‘get property of’ block is used to get the coordinates of the treasure (which are available only on this level.)


Simon 705 132 Back

Run the program and watch what happens. Can you finish the level?

```
when [flag] clicked
go to x: (x-position of Treasure) y: (y-position of Treasure)
get treasure!
```

Figure 9.4: Screenshot of the finished solution to the Pirate Plunder ‘get property of’ tutorial added for Study 4

Help Function

The main feedback from players in Study 3 was that the hint function was unhelpful because it gave the same hint throughout the level. For Study 4, a full help function was added that contains between three and six 'scaffolded' steps for each challenge. These were written based on teacher and researcher observations in Study 3, including regularly given responses to player questions and suggestions for how to complete the level. Most steps were framed as questions because the game cannot detect whether the player has already completed the suggested step. For the last step, the player is asked if they want to do the related tutorial again. For example, the help suggestions for the first custom block challenge (level 19) are as follows, with an explanation of why that suggestion is given under each:

1. "Take a look at the grid, what tasks can we repeat?"

Recognising the duplicated functionality is a key part of creating the correct custom block(s) for each level. This first suggestion is used in all of the custom block and input challenges.

2. "We repeat 4 move 1 and turn left. We do this 3 times."

Building on the last suggestion, indirectly telling the player what should go inside the custom block.

3. "We can use a custom block for those repeated actions."

Reminding the player that they should be using custom blocks for those repeated actions (shown in Figure 9.5).

4. "Have you added blocks to the 'define' block?" (pointer to the first define block in the program.)

This was an issue observed when players used custom blocks in Study 3. Players would not add blocks to their define blocks in early custom block challenges. Instead, they would expect that the block would achieve the required functionality by naming the define block what they wanted it to do and not adding blocks to it.

5. "You can use your block from the 'More Blocks' folder" (pointer to the 'More Blocks' category.)

Reminding the player that once they have created a custom block, they need to use it from the 'More Blocks' folder.

6. "Do you want to do the custom blocks tutorials again?" (with an option to return to the level select screen.)

Telling the player to go and do the custom blocks tutorial again if the previous suggestions did not help them complete the level.

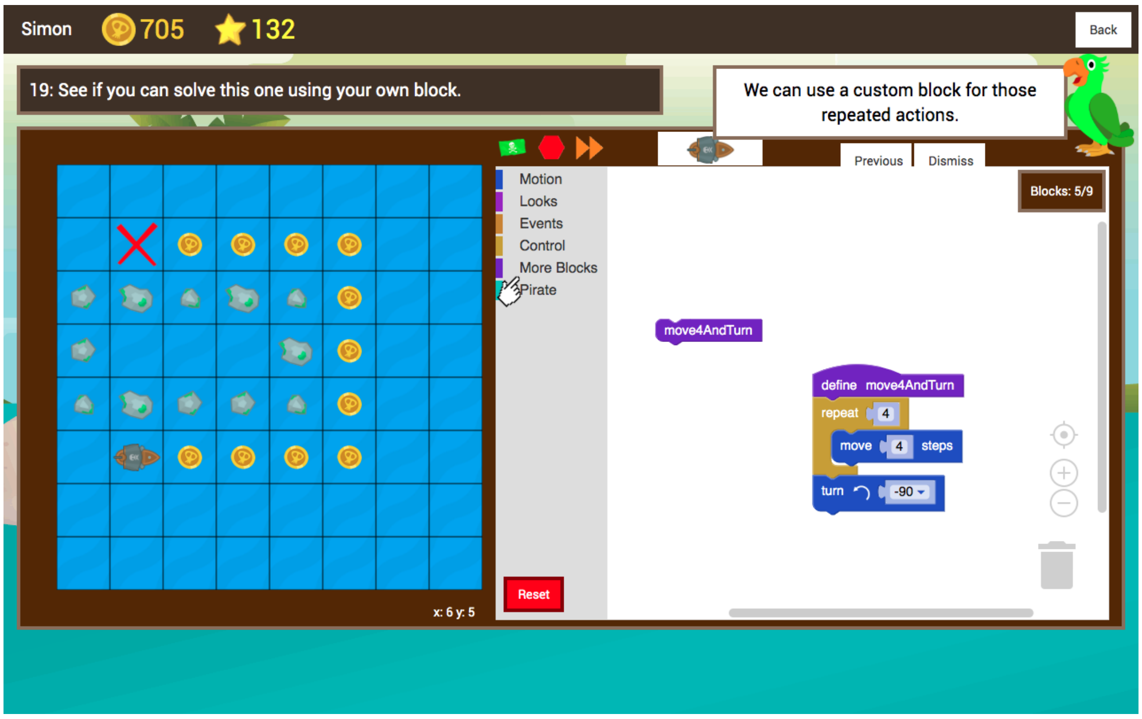


Figure 9.5: Screenshot of the Pirate Plunder help function

To access the help, the player clicks on the green feedback parrot in the top-right corner of a challenge level. The parrot may then move around the screen or a pointer may appear depending on the suggestion. This is similar to the tutorial levels described in Section 7.4.5. When a help suggestion is visible, the player can either dismiss it or return to the previous suggestion. The next suggestion can only be accessed by clicking on the parrot again. This was done in an attempt to stagger the help, so the player does not cycle through all the suggestions in one go.

Closer to Scratch

During the artifact-based interviews in Study 3, several participants stated that Scratch was too different from Pirate Plunder and that this affected their solution to the post-test Scratch task. For Study 4, three changes were made to Pirate Plunder to make the user interface more similar to Scratch. These were 1) the removal of the x and y-axes from the grid, 2) moving the coordinate position indicator below the grid and 3) using sprite images instead of names (all visible in Figure 9.5).

One frequent observation from the post-test Scratch assessment in Study 3 was that participants struggled with coordinate positions. This is because Scratch does not have the coordinates on the grid axis (like the Study 3 version of Pirate Plunder) and requires the use of the coordinate position indicator to get sprite coordinates. For Study 4, the x and y-axes were removed from Pirate Plunder and the coordinate position indicator moved to below the grid, where it is in Scratch.

Pirate Names

The Study 3 version of Pirate Plunder used participants' first names as player identifiers. However, with the introduction of the General Data Protection Regulation (GDPR) laws, it was decided that participant names should be removed as they could lead to player identification. This resulted in player name customisation being introduced for Study 4.

Players are asked to choose a starting avatar when they first log in to the game. For Study 4, this was extended to require the player to input their pirate name (Figure 9.6), which can then be updated in the shop. Players were told not to use their real names. The name is shown on the class and login screen and can be used by players to identify each other. This is similar to the username systems used on Xbox Live and the PlayStation Network. As the player can update their name as many times as want, player ID numbers were added to the login screen so that players could be identified should they forget which avatar is theirs. In addition, a language filter was added to the name input to recognise and block profanities.

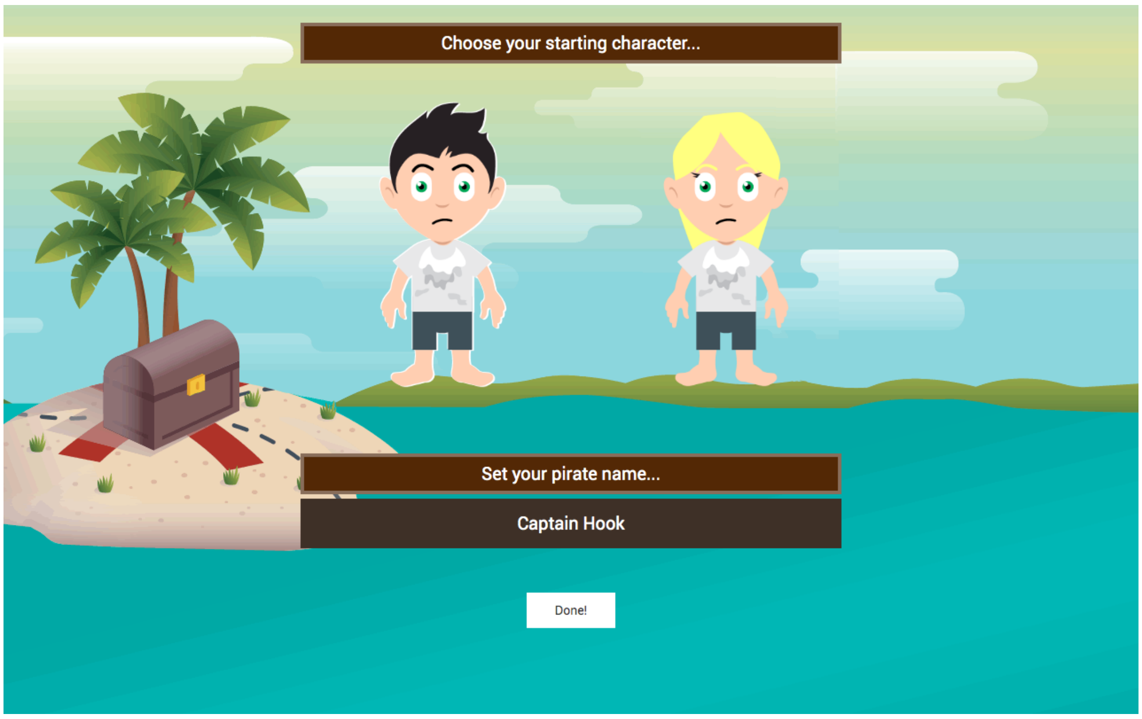


Figure 9.6: Screenshot of the Pirate Plunder character select screen for Study 4 where the player can set their pirate name

Other Changes

Several other minor changes were made to Pirate Plunder for Study 4. These included removing the five-coin charge to update avatars in the shop, adding additional shop items that were requested from the feedback in Study 3 and adding additional analytics for help and name changes.

9.1.3 Assessment Changes

The two main hypotheses for not finding a difference in Study 3 were both related to the Scratch assessment (Section 8.5). The first was that participants were not motivated to use abstraction in the task because there was no block limit or obvious duplication like when they encountered it in Pirate Plunder. In addition, there would only be duplication (the rationale for using abstraction) if the player had produced it, meaning that they would have to complete the task first and then refactor their code using abstraction. This requires a lot of work for participants to produce in the 40 minutes they were given. The second was that because participants were doing the same task at pre-and post-test, they could complete it at post-test by reproducing or expanding their pre-test solution, bypassing what they had learnt in Pirate Plunder. These observations resulted in a new Scratch assessment designed to deal with these issues and an additional multiple-choice assessment on abstraction in Scratch. These are both described in the materials section (Section 9.3.3).

9.1.4 Experimental Design Changes

Study 3 was designed primarily to compare the debugging-first and non-debugging versions of Pirate Plunder. Therefore, it was limited when trying to establish whether Pirate Plunder was effective in teaching abstraction in Scratch. The lack of a comparison to a non-programming control group (no block-based programming) meant that the results of the CT test were not useful as both groups had been practising CT during the study.

To address these issues, Study 4 used a partial-crossover design with a non-programming control, in addition to the Scratch control used in Study 3. All participants play Pirate Plunder, meaning that it is easier to establish whether the game meets the learning outcomes. The new design is described in Section 9.3.2.

9.2 Pilot Study

The pilot study took place before the start of Study 4 in the same medium-sized primary school as the Study 3 pilot. The participants were nine of the 12 children who had taken part in the earlier pilot, who were now age 10 and 11. They were experienced in both Scratch and Pirate Plunder, having played a previous version of the game for 5 hours during the Study 3 pilot.

The pilot took place over two days and involved 2 hours 15 minutes of total activity. They first spent 1 hour 30 minutes playing Pirate Plunder. Participants started from the custom block tutorials to fit the time constraints and because they already had a basic understanding of the game. This meant that the assessment tasks could be more effectively tested as they require knowledge of custom blocks and cloning. The participants then had 30 minutes to attempt the Scratch challenge and 15 minutes to do the multiple-choice Scratch abstraction test. How this influenced the assessment tasks is discussed in Section 9.3.3.

9.3 Method

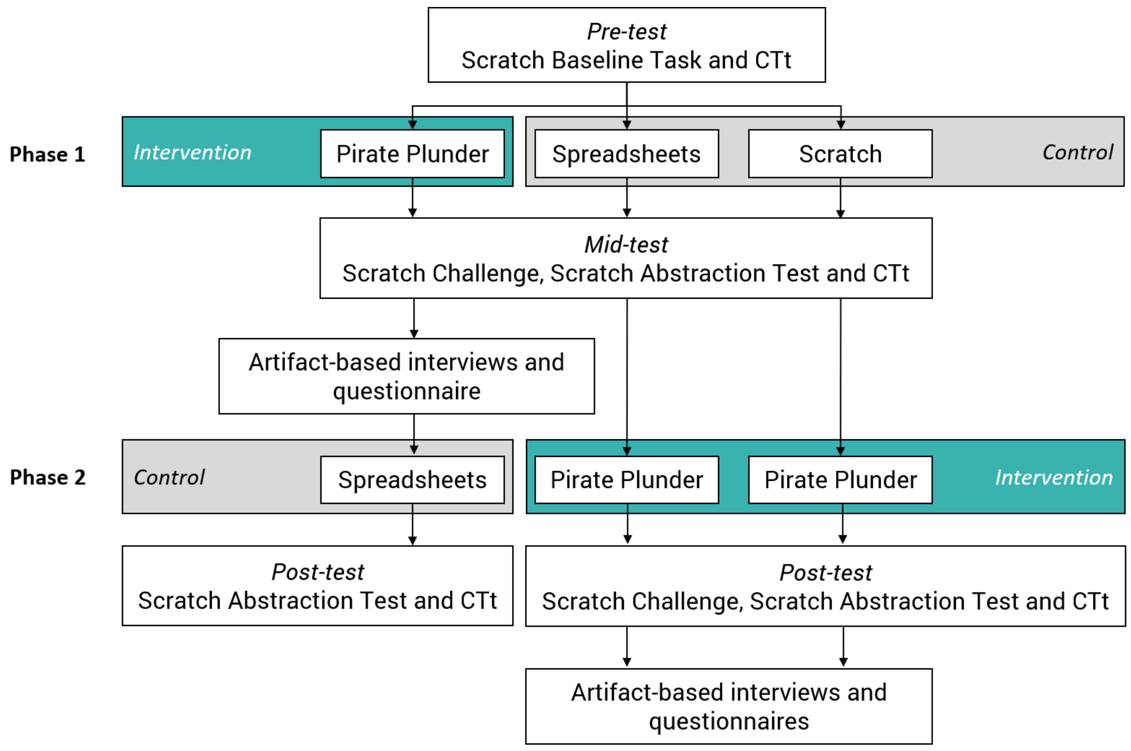
9.3.1 Participants

The participants in Study 4 were 91 children age 10 and 11 (*M* = 10.58, *SD* = 0.32) from a large primary school in northern England (the same school as Study 3 but with the next cohort). As with Study 3, they were largely inexperienced with Scratch (having had sporadic lessons throughout primary school) and had no experience using custom blocks or cloning. The sample contained 45 males (49.5%) and 46 females (50.5%).

9.3.2 Design

Study 4 followed a pre-test post-test partial-crossover quasi-experimental design to measure for improvements using abstraction in Scratch after playing Pirate Plunder (Figure 9.7). For the first part of the study, the three groups were split into Pirate Plunder (intervention), spreadsheets (non-programming active control) and Scratch (programming active control). The two control groups then crossed-over to Pirate Plunder and the intervention group to the spreadsheets curriculum. Participants were assessed for their Scratch baseline ability using the Scratch task from Study 3, their ability to use abstraction in Scratch through the Scratch challenge and the multiple-choice Scratch abstraction test, and their computational thinking ability using the Computational Thinking test. After playing Pirate Plunder, participants were given confidence questionnaires and a sample of them were interviewed.

The study uses a partial-crossover (as opposed to a full crossover with each group doing each task) partially because of school timetabling and planning restraints, but mainly because it met the aims of the study without participants completing the assessments four times each. Pirate Plunder can be compared against the non-programming and programming curricula in Phase 1 and the crossover allows all participants to play the game. This goes some way to handling the effects of using class groups (which was done again in this study to reduce logistical issues for the school) as the assessments are completed after each group has played the game. However, because it is a partial-crossover, Phase 2 is not as informative because it does not contain a genuine control group (as they have already played the game).



Pre-test
Scratch Baseline Task and CTt
Phase 1
Intervention
Pirate Plunder
Spreadsheets
Scratch
Control
Mid-test
Scratch Challenge, Scratch Abstraction Test and CTt
Artifact-based interviews and questionnaire
Phase 2
Control
Spreadsheets
Pirate Plunder
Pirate Plunder
Intervention
Post-test
Scratch Abstraction Test and CTt
Post-test
Scratch Challenge, Scratch Abstraction Test and CTt
Artifact-based interviews and questionnaires

Figure 9.7: Diagram of the Study 4 design

9.3.3 Materials

Spreadsheets Curriculum

A six-lesson spreadsheets curriculum was chosen as the primary control group activity for this study because it does not involve programming (beyond using pre-made formula to calculate values, e.g. SUM and AVERAGE) or explicit CT, yet still has participants using the computers and being exposed to new learning content. As with the Scratch curriculum, the spreadsheets curriculum was produced by Twinkl (2018) and is designed for the age group (age 10 and 11). Table 9.1 gives a breakdown of each lesson.

Both curricula in this study were delivered by the author and were given the same amount of time as Pirate Plunder.

Scratch: Animated Stories Curriculum

The same six-lesson Scratch curriculum from Study 3 was used and is described in detail in Section 8.3.3. It involves creating an animated story in Scratch based on a ‘haunted house’ starter project. The curriculum contains no custom blocks or cloning.

Scratch Baseline Task

The Scratch task assessment from Study 3 was used as a baseline of Scratch proficiency at pre-test for this study. As described in Section 8.3.3, the task involves

Table 9.1: Spreadsheets - lesson breakdown

Lesson Number	Lesson Name	Content
1	Number Operations	Enter and edit text and numbers in cells and use SUM formula, begin formatting cells.
2	Ordering and Presenting Data	Using SUM formula for a specific purpose, ordering data using the sort function and producing graphs to present data.
3	Add, Edit and Calculate Data	Creating totals and averages on existing data, sorting and understanding the benefit of automatic recalculation.
4	Solving Problems	Investigating how to use a spreadsheet to solve a given problem.
5	Party Plan Budget	Choosing items for a party from a list of possible items and prices, using a spreadsheet to calculate quantities and totals within a set budget for a given number of people.
6	Design Your Own	Open-ended challenge to design their own spreadsheet.

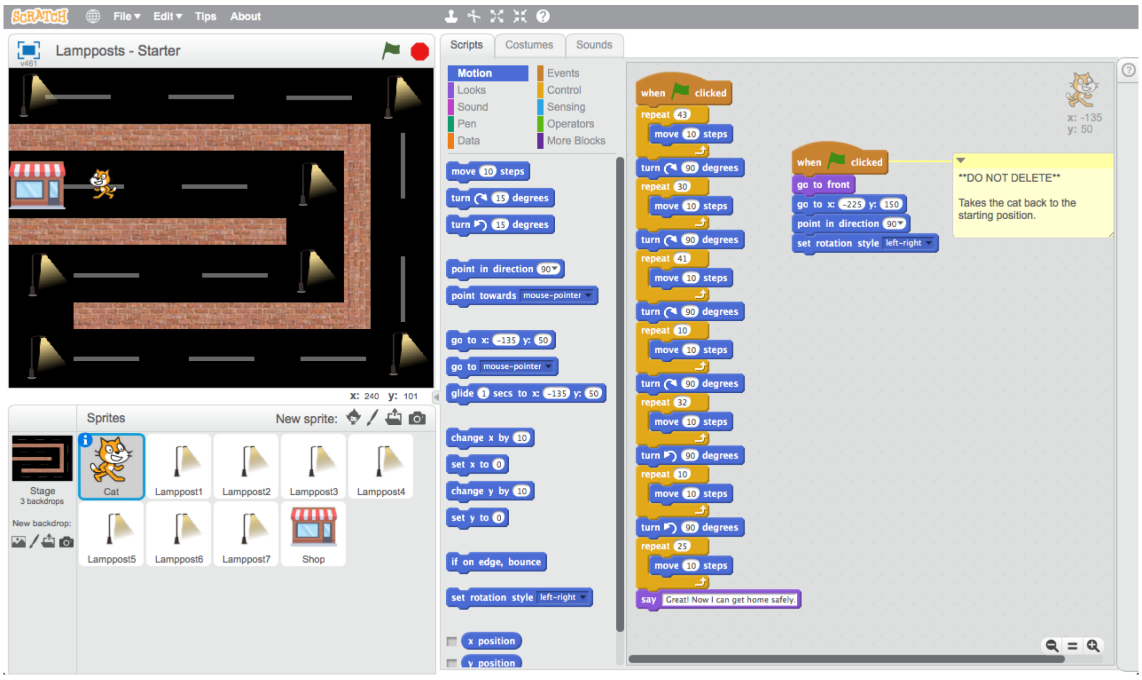
animating the Scratch cat around the edges of a rectangle and leaving an object on each corner. Two versions were used alternately for participants sat next to each other in an attempt to reduce copying. The task was analysed using Dr. Scratch for abstraction and decomposition and the expected outcome measure in Table 8.3.

This was used at pre-test instead of the Scratch challenge assessment because it enables participants to achieve an outcome without prior Scratch knowledge. Whereas the Scratch challenge requires specific functionality that none of the participants had previously encountered. Furthermore, using the Scratch challenge at pre-test would have meant that the Phase 1 control groups would have done the challenge twice before they had been taught how to attempt it using abstraction.

Scratch Challenge

The Scratch challenge assessment was created using the conclusions from Study 3 summarised in Section 9.1.1. Participants were instructed to reduce the block count in a pre-made project that contained both duplicated blocks and sprites, therefore giving them motivation for using abstraction.

The project involves animating the Scratch cat around a map and leaving a lamp-post sprite on each corner (Figure 9.8). Participants were specifically instructed to reduce the number of blocks and sprites used in the starter project. The ideal solution uses custom blocks and cloning as seen in Figure 9.9. The task sheet (Appendix Q) contains screenshots of the starting blocks on the reverse so that the player can recreate the starting program or use the input values if needed. This was done because of observations during the pilot study, where participants removed blocks from the program and were unable to remember the distance values that they needed to move the cat sprite.



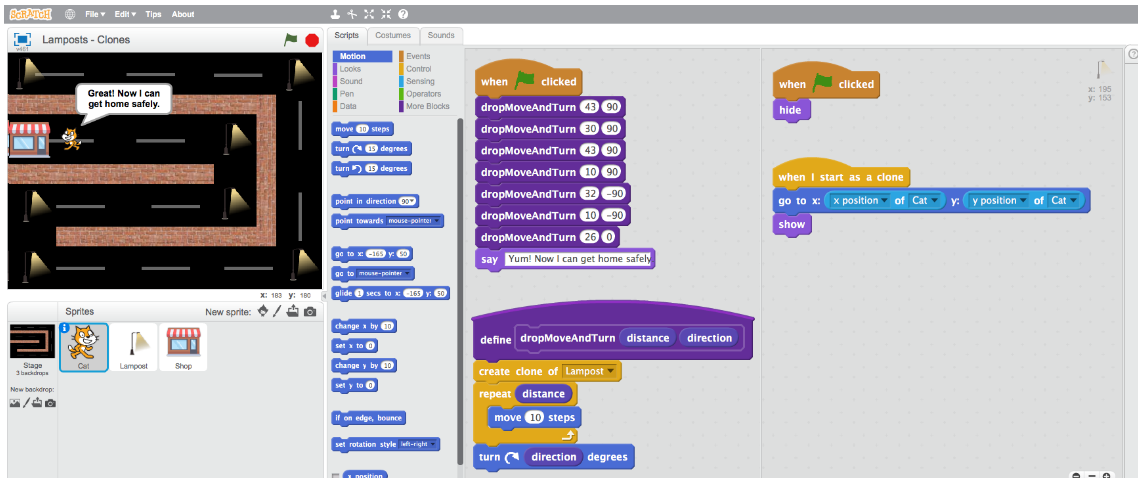
```
when [green flag] clicked
repeat 43
    move 10 steps
turn ↻ 90 degrees
repeat 30
    move 10 steps
turn ↻ 90 degrees
repeat 41
    move 10 steps
turn ↻ 90 degrees
repeat 10
    move 10 steps
turn ↻ 90 degrees
repeat 32
    move 10 steps
turn ↺ 90 degrees
repeat 10
    move 10 steps
turn ↺ 90 degrees
repeat 25
    move 10 steps
say Great! Now I can get home safely.
```

```
when [green flag] clicked
go to front
go to x: -225 y: 150
point in direction 90
set rotation style left-right
```

DO NOT DELETE

Takes the cat back to the starting position.

Figure 9.8: Screenshot of the Scratch challenge starter project (the script in the right is normally off the screen and out of the immediate view of the participant)



```
when [green flag] clicked
dropMoveAndTurn 43 90
dropMoveAndTurn 30 90
dropMoveAndTurn 43 90
dropMoveAndTurn 10 90
dropMoveAndTurn 32 -90
dropMoveAndTurn 10 -90
dropMoveAndTurn 26 0
say Yum! Now I can get home safely.

define dropMoveAndTurn (distance) (direction)
create clone of Lampost
repeat (distance)
    move 10 steps
turn ↻ (direction) degrees
```

```
when [green flag] clicked
hide

when I start as a clone
go to x: (x position of Cat) y: (y position of Cat)
show
```

Figure 9.9: Screenshot of the perfect solution to the Scratch challenge, showing the cat sprite (left) and lamppost sprite (right)

In the Study 3 Scratch task, it was observed that participants often removed the starting blocks (used to reset the cat sprite position). To deal with this, these blocks were added to a separate ‘when green flag clicked’ block off to the right-hand side of the project. Participants were then instructed not to delete them by the task sheet. They were told they could not use ‘glide to position’ or ‘go to position’ blocks that allow movement to be reduced from the two blocks of ‘repeat’ and ‘move’ to a single block (therefore easily reducing the block count).

As with the Scratch baseline task, each project was analysed using Dr. Scratch for abstraction and decomposition (Table 8.2). The starter project gets 1 point for using multiple scripts in multiple sprites. Projects were also manually analysed against the following criteria (an explanation of each is given below) as a measure of whether abstraction had been used correctly. This is because Dr. Scratch can only measure whether a block has been used in a project, not whether it has been used correctly.

Assessment criteria:

1. Correct custom block

A custom block with two inputs representing distance and degrees (may not be named correctly), containing a 'repeat' 'move' for the distance and a 'turn' for the direction.

2. Correct use of cloning

A single lamppost sprite that is cloned at the position of the cat sprite (using the 'get property of' block) inside the custom block before the repeat (because the starter project has a lamppost at the starting position.)

3. Complete movement

The cat sprite is animated around the map and reaches the shop as it does in the starter project.

4. Correct lamppost positions

All the lamppost sprites are in the same positions as they are in the starter project. This includes the lamppost on the cat's starting position and on each subsequent corner. They must appear in sequence (ideally as the cat sprite reaches them.)

Multiple-Choice Scratch Abstraction Test

The Scratch abstraction test was a 10-question multiple-choice assessment designed by the author and used to supplement the Scratch challenge assessment. The questions are on using custom blocks and cloning correctly in Scratch (the full test is in Appendix R and the breakdown and rationale for each question is shown in Appendix S). Each question has four options with one correct answer. The test includes questions on:

- Identifying duplicated Scratch code that can be refactored using a custom block.
- Identifying correct block names and inputs for duplicated code.
- Comparing Scratch scenes and figuring out which sprites can be cloned.
- Identifying the block that can be used to get properties of a sprite.
- Identifying the blocks used to clone sprites successfully.

Computational Thinking Test

As in Study 3, the Computational Thinking test (CTt) was used as a measure of CT. The CTt is a 28-question multiple-choice assessment that uses visual arrows and blocks common in education programming tools (Román-González et al., 2016) and was used because it is one of the better researched CT measures.

Questionnaires

The Pirate Plunder questionnaire from Study 3 (Appendix M) was given to participants after they had played the game. This included questions about confidence using Scratch, custom blocks and cloning, as well as the participant's change in confidence after playing Pirate Plunder. As in Study 3, answers were coded as quantitative data: confidence questions as 3 (very confident), 2 (confident), 1 (slightly confident) and 0 (not confident) and confidence change questions as 1 (improved), 0 (same as before) and -1 (declined).

Artifact-Based Interviews

Artifact-based interviews were conducted for the Pirate Plunder groups after they had played the game, using the same script as Study 3 (Appendix L). This was done to establish whether participants had understood the rationale for using abstraction in Scratch. Interviews began with open questions about the participant's project, such as what each of the blocks did and why they had used them. Before asking more leading questions about their use (or lack) of custom blocks and cloning in the project and where they could use them in other Scratch projects.

9.3.4 Procedure

All participants did the Scratch baseline task and the CTt at pre-test. The Scratch baseline task took place in the school IT suite in class groups. Participants were introduced to the study and the assessment task. They were then given 40 minutes to produce a Scratch project to the assessment specification. The CTt was administered using tablets in the classroom after the group had completed the Scratch baseline task. Participants were given a maximum of 45 minutes to complete the test.

Class groups were then assigned to the intervention (Pirate Plunder) or active control conditions (spreadsheets and Scratch) for Phase 1 of the study. Both phases were four weeks long with two sessions per week, taking place in the school IT suite. Participants in the intervention group played the game for that time with no classroom support other than individual assistance (if required). The six-lesson spreadsheet and Scratch curricula were delivered by the author over the four weeks.

At mid-test, all participants did the Scratch challenge (as opposed to the Scratch baseline task), multiple-choice Scratch abstraction test and the CTt. The intervention group also completed the Pirate Plunder questionnaire and a sample of them were interviewed. Once again, the Scratch challenge took place in the IT suite with participants given 40 minutes to modify the starter project. Both the multiple-choice Scratch abstraction test and CTt (in that order) were then administered using tablets in the classroom. They were given a maximum of 15 minutes for the abstraction test and 45 minutes for the CTt.

The conditions were then crossed-over so that the intervention group did spreadsheets and the two control groups from the first Phase did Pirate Plunder (Figure 9.7). At post-test, the intervention groups re-completed the Scratch challenge, multiple-choice Scratch abstraction test and the CTt, whilst the control group only did the abstraction test and the CTt.

For the interviews at mid-test and post-test, 15 participants from each group (*N* = 45) were interviewed after completing their post-Pirate Plunder Scratch challenge. To select participants for the interviews, each group was divided into three categories (using simpler criteria than the ones given in Section 9.4.4): correct solution, almost correct or interesting solution and no use of abstraction. Depending on the group, roughly five participants who met each criterion were selected.

The study took place in the first term of the school year, before the Christmas break. It was 10 weeks long, four weeks for the two phases (two sessions per week, 30 minutes and 50 minutes respectively), with a week for the mid-test and interviews in the middle and one at the end for the post-test and further interviews. The pre-test was done on the Monday of the first study week to fit with the school time constraints. It started on the 8th October 2018 and finished on the 19th December 2018.

9.3.5 Ethics and Access to Participants

The study falls under the same ethics approval as Study 3. For this study, permission was obtained from the headteacher to confirm that the study could go ahead. After meeting with the class teachers to confirm the study content, opt-in consent forms were sent out to the parents/guardians of potential participants (Appendix T) along with an information sheet with a short description of the study and the data collected (Appendix U). The study went ahead once the permission slips had been returned.

As with Study 3, all data was anonymised and stored against the participant ID numbers. The data management plan is shown in Appendix H.

9.3.6 Hypotheses

There were two hypotheses for the study, one related to abstraction skills and the other to CT ability:

1. Pirate Plunder would improve scores on abstraction measures from pre-to post-test in comparison with both the programming (Scratch) and non-programming (spreadsheets) control groups.

2. Pirate Plunder would improve CT ability (measured by the CTt) in comparison to the non-programming control group who were not doing explicit CT activities.

9.4 Results

9.4.1 Phase 1

Phase 1 of the study was from pre-to mid-test, comparing Pirate Plunder with both non-programming (spreadsheets) and programming (Scratch) conditions.

Hypothesis 1 - Did Pirate Plunder Improve Abstraction in Scratch?

Hypothesis 1 was that the intervention group would improve on the abstraction measures compared to the control groups. This includes the Dr. Scratch abstraction and decomposition scores on the Scratch challenge assessment and the multiple-choice Scratch abstraction test scores.

For this phase, the mid-test Scratch challenge abstraction scores for each group are compared using the Scratch baseline task abstraction scores as a covariate, to control for variance in baseline ability. Figure 9.10 shows the mean Dr. Scratch abstraction and decomposition for each group at mid-test (note that the starting project gets 1 point for abstraction in Dr. Scratch). A one-way ANCOVA showed a significant difference in abstraction scores between the three groups, $F(2, 78) = 30.30$, $p < .001$, $\eta^2 = .44$. Table 9.2 shows the average Dr. Scratch abstraction scores for each group. Table 9.3 gives the breakdown of these results for each group, showing how many participants used custom blocks and how many used cloning.

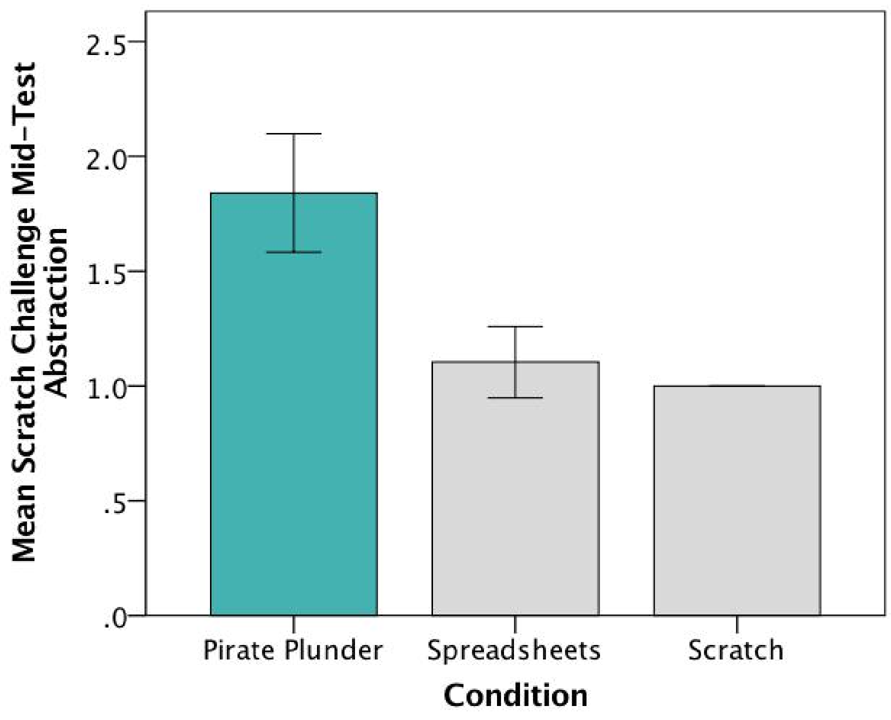


Figure 9.10: Comparison of the Dr. Scratch abstraction and decomposition scores on the mid-test Scratch challenge for each group (error bars show 95% confidence interval)

Table 9.2: Descriptive statistics of the Dr. Scratch abstraction and decomposition scores on the Scratch task and Scratch challenge for each group (maximum score of 3)

Condition		Scratch Baseline Task (Pre-Test)	Scratch Challenge (Mid-Test)
	M	1.08	1.84
Pirate Plunder	*N*	25	25
	SD	0.70	0.62
	M	0.93	1.10
Spreadsheets	*N*	29	29
	SD	0.37	0.41
	M	1.11	1.00
Scratch	*N*	28	28
	SD	0.40	0.00

Table 9.3: Breakdown of the Dr. Scratch abstraction and decomposition scores on the mid-test Scratch challenge for each group

Condition	*N*	Dr. Scratch Abstraction and Decomposition Score			
		0	1	2	3
Pirate Plunder	25	0	7	15	3
Spreadsheets	29	0	27	1	1
Scratch	28	0	28	0	0
Total	82	0	62	16	4

Figure 9.11 shows the mean multiple-choice Scratch abstraction test scores at mid-test for each group. There was a significant difference between the three groups using a one-way ANOVA ($F(2, 80) = 11.64$, $p < .001$, $\eta^2 = .23$), with the Pirate Plunder group ($M = 5.21$, $N = 28$, $SD = 1.40$) scoring higher than both the non-programming ($M = 3.58$, $N = 26$, $SD = 1.86$) and programming control ($M = 3.45$, $N = 29$, $SD = 1.30$).

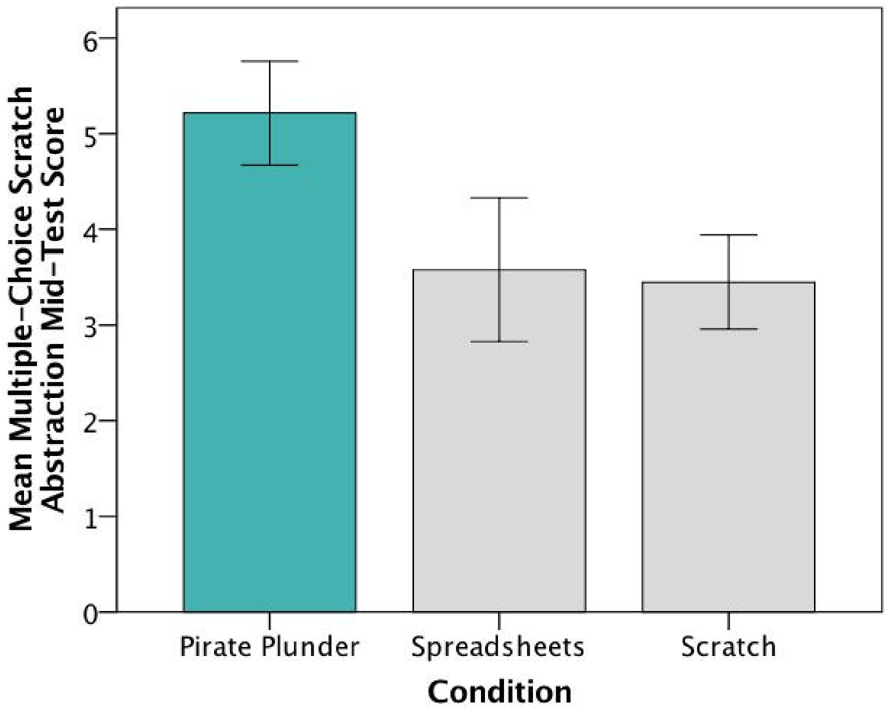


Figure 9.11: Comparison of the mid-test multiple-choice Scratch abstraction test scores for each group (error bars show 95% confidence interval)

Hypothesis 2 - Did Pirate Plunder Improve Computational Thinking?

Hypothesis 2 was that participants in the intervention group would improve their CT ability (measured by the CTt) in comparison with the non-programming (spreadsheets) control group.

There was a significant difference in the mean CTt learning gains from pre-to mid-test between the three groups (*F*(2, 84) = 3.72, *p* = .028, η^2 = .081) (Figure 9.12), with the only significant pairwise-comparison (using an independent samples t-test) between the intervention group and the non-programming control, *t*(55) = 2.87, *p* = .015, *d* = 0.67 (Table 9.4).

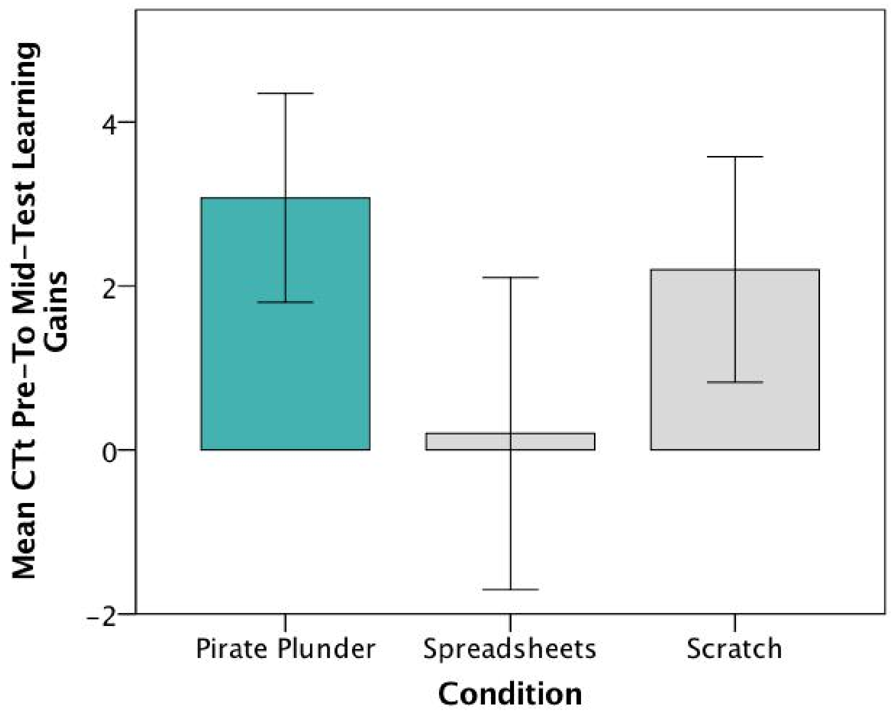


Figure 9.12: Comparison of the CTt learning gains from pre-to mid-test for each group (error bars show 95% confidence interval)

Table 9.4: Descriptive statistics of the Computational Thinking test from pre-to mid-test for each group (maximum score of 28)

Condition		Pre-Test	Mid-Test	Learning Gains
	M	14.26	17.33	3.07
Pirate Plunder	*N*	27	27	27
	SD	5.90	5.61	3.22
	M	14.33	14.53	0.20
Spreadsheets	*N*	30	30	30
	SD	5.00	5.44	5.10
	M	15.70	17.90	2.20
Scratch	*N*	30	30	30
	SD	4.33	3.94	3.68

9.4.2 Phase 2

Phase 2 of the study was from mid-to post-test. During this phase, the Phase 1 intervention group switched to a non-programming control and both Phase 1 control groups switched to Pirate Plunder (Figure 9.7). This means that there was not a genuine control group during Phase 2. Each group's activities are shown in this section using Phase 1/Phase 2 identifiers (e.g. Pirate Plunder/Spreadsheets).

Hypothesis 1 - Did Pirate Plunder Improve Abstraction in Scratch?

Table 9.5 shows the learning gains on the Scratch challenge for the Phase 2 intervention groups. The breakdown of these results is given in Table 9.6, showing how many participants used custom blocks and cloning. Both groups improved significantly from mid-to post-test: Spreadsheets/Pirate Plunder ($t(28) = 5.52$, $p < .001$, $d = 1.44$) and Scratch/Pirate Plunder ($t(25) = 8.76$, $p < .001$, $d = 1.44$). The scores are not compared with the control group because they did not re-complete the assessment at post-test.

Table 9.5: Descriptive statistics of the Dr. Scratch abstraction and decomposition scores on the Scratch challenge from mid-to post-test for each group (maximum score of 3)

Condition		Mid-Test	Post-Test	Learning Gains
	M	1.10	1.90	0.79
Spreadsheets/Pirate Plunder	*N*	29	29	29
	SD	0.41	0.67	0.77
	M	1.00	2.19	1.19
Scratch/Pirate Plunder	*N*	26	26	26
	SD	0.00	0.69	0.69

Table 9.6: Breakdown of the Dr. Scratch abstraction and decomposition scores on the post-test Scratch challenge for the Phase 2 intervention groups

Condition	*N*	Dr. Scratch Abstraction and Decomposition Score			
		0	1	2	3
Spreadsheets/Pirate Plunder	29	0	8	16	5
Scratch/Pirate Plunder	26	0	4	13	9

Figure 9.13 shows the mid-to post-test learning gains for the multiple-choice Scratch abstraction test for each group. A one-way ANOVA showed that there was no significant difference between groups, $F(2, 71) = 2.21$, $p = .12$, $\eta^2 = .059$. In addition, using a paired samples t-test, only the Scratch/Pirate Plunder group changed (in this case, improved) significantly from mid-to post-test, $t(26) = 2.14$, $p = .042$, $d = 0.47$. The descriptive statistics are shown in Table 9.7.

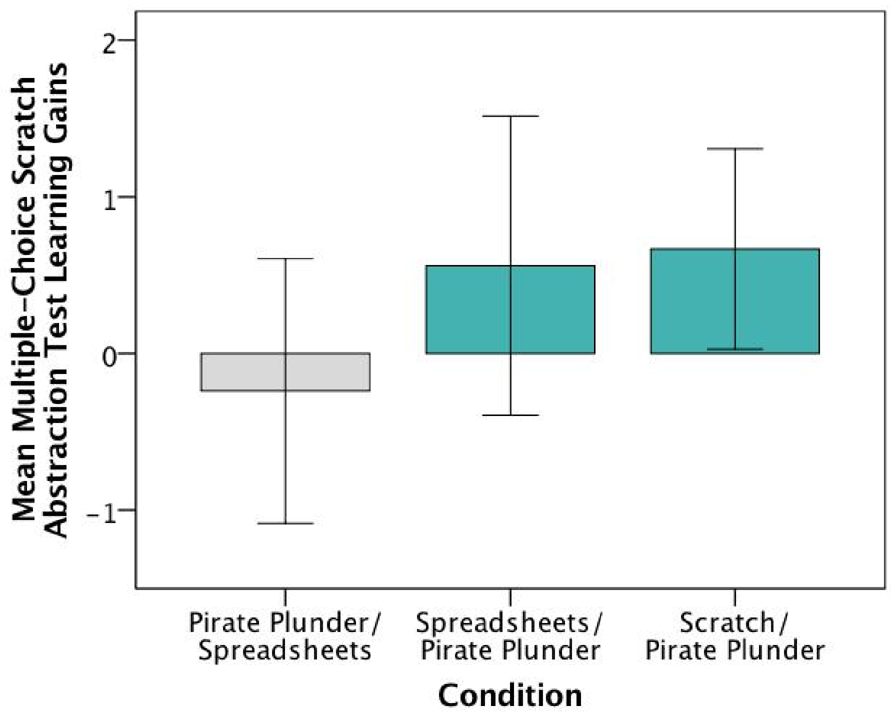


Figure 9.13: Comparison of the multiple-choice Scratch abstraction test learning gains from mid-to post-test for each group (error bars show 95% confidence interval)

Table 9.7: Descriptive statistics for the multiple-choice Scratch abstraction test scores from mid-to post-test for each group

Condition		Mid-Test	Post-Test	Learning Gains
	M	5.12	4.88	-0.24
Pirate Plunder/Spreadsheets	*N*	25	25	25
	SD	1.45	2.05	2.05
	M	3.64	4.20	0.56
Spreadsheets/Pirate Plunder	*N*	25	25	25
	SD	1.87	1.87	2.31
	M	3.41	4.07	0.67
Scratch/Pirate Plunder	*N*	27	27	27
	SD	1.31	1.49	1.62

Hypothesis 2 - Did Pirate Plunder Improve Computational Thinking?

Figure 9.14 shows the learning gains from mid-to post-test on the CTt for each group. A one-way ANOVA showed a significant difference between groups, *F*(2, 84) = 4.49, *p* = .014, η^2 = .097. A paired samples t-test showed that the Pirate Plunder/Spreadsheets group declined significantly, *t*(27) = 2.87, *p* = .008, *d* = 0.38. Table 9.8 shows the descriptive statistics.

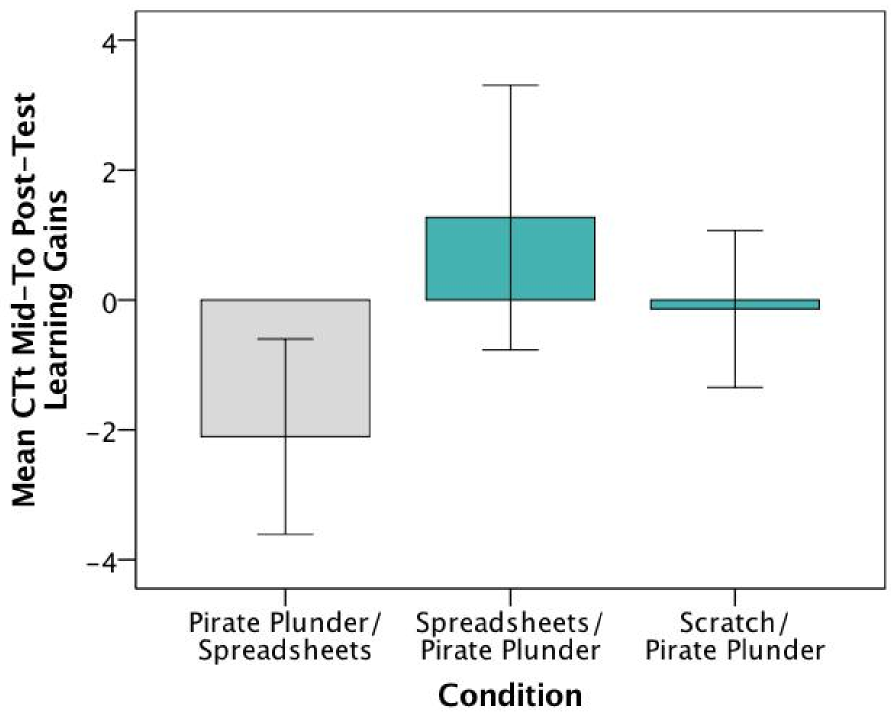


Figure 9.14: Comparison of the CTt learning gains from mid-to post-test for each group (error bars show 95% confidence interval)

Table 9.8: Descriptive statistics of the Computational Thinking test from mid-to post-test for each group (maximum score of 28)

Condition		Mid-Test	Post-Test	Learning Gains
	M	17.29	15.18	-2.11
Pirate Plunder/Spreadsheets	*N*	28	28	28
	SD	5.51	5.64	3.88
	M	14.53	15.80	1.27
Spreadsheets/Pirate Plunder	*N*	30	30	30
	SD	5.44	5.90	5.46
	M	18.07	17.93	-0.14
Scratch/Pirate Plunder	*N*	29	29	29
	SD	3.90	5.01	3.17

9.4.3 Pirate Plunder Performance

Table 9.9 shows the Pirate Plunder performance for each group. As with Study 3, this is judged by the number of challenges completed (out of 40) and overall stars collected (maximum of 120). The average stars collected (out of 3) and the time spent playing the game are also given. One-way ANOVAs showed no significant difference between the three groups for challenges completed ($F(2, 87) = 0.81$, $p = .447$, $\eta^2 = .018$) or stars collected ($F(2, 87) = 1.13$, $p = .329$, $\eta^2 = .025$).

Table 9.9: Descriptive statistics of Pirate Plunder performance for each group

Condition		Challenges Completed	Total Stars Collected	Average Stars per Level	Total Time (Minutes)
Pirate Plunder/Spreadsheets	*M*	33.00	95.72	2.89	316
	N	29	29	29	
	SD	6.51	21.84	0.21	
Spreadsheets/Pirate Plunder	*M*	32.23	94.90	2.94	321
	N	30	30	30	
	SD	5.93	18.73	0.10	
Scratch/Pirate Plunder	*M*	34.13	101.52	2.97	326
	N	31	31	31	
	SD	5.05	15.40	0.05	

9.4.4 Completeness of Scratch Challenge Solution After Playing Pirate Plunder

Completeness criteria for the Scratch challenge are used to support the Dr. Scratch abstraction and decomposition scores. This is because, as mentioned earlier in the chapter, Dr. Scratch can only assess whether a block has been used within the project and not whether the block has been used correctly. Each project was assessed using the four criteria explained in Section 9.3.3:

1. Correct custom block
2. Correct use of cloning
3. Complete movement
4. Correct lamppost positions

Table 9.10 shows the number of participants that met each criterion in their Scratch challenge projects at mid-and post-test (note that the Pirate Plunder/Spreadsheets group did not repeat the Scratch challenge assessment at post-test). When combined with the Dr. Scratch abstraction and decomposition scores, this provides a measure of how successful participants were in using abstraction. There were significant improvements for the Phase 2 intervention groups from mid-to post-test in using the correct custom block (Spreadsheets/Pirate Plunder, $t(28) = 6.84$, $p < .001$, $d = 1.80$ and Scratch/Pirate Plunder, $t(25) = 9.21$, $p < .001$, $d = 2.56$) and the correct use of cloning (Spreadsheets/Pirate Plunder, $t(28) = 2.42$, $p = .023$, $d = 0.63$ and Scratch/Pirate Plunder, $t(25) = 2.13$, $p = .043$, $d = 0.58$). This supports the improvements shown in the Dr. Scratch abstraction and decomposition scores and shows that participants were not only using abstraction but were using it successfully.

Table 9.10: Descriptive statistics of the completeness of the post-Pirate Plunder Scratch challenge projects (PP/SP = Pirate Plunder/Spreadsheets, SP/PP = Spreadsheets/Pirate Plunder and SC/PP = Scratch/Pirate Plunder)

Group	*N*		Correct Custom Block		Correct Use of Cloning		Complete Movement		Correct Lamppost Positions	
	Mid	Post	Mid	Post	Mid	Post	Mid	Post	Mid	Post
PP/SP	25	-	10	-	0	-	16	-	21	-
SP/PP	29	29	0	11	0	5	9	19	22	26
SC/PP	26	26	0	13	0	4	8	22	9	15

9.4.5 Artifact-Based Interview Observations

The purpose of the artifact-based interviews was to investigate whether participants had understood the underlying rationale for using abstraction in Scratch projects. To establish this, they were asked why they had used/would use custom blocks and cloning and to give other examples of where they could use each in Scratch (Section 8.3.3). The answers were roughly split into three categories, where the participant had either:

- Understood abstraction and could apply it to examples outside of Pirate Plunder.
- Understood abstraction but could only apply it to Pirate Plunder-based examples (i.e. involving moving, turning and cannonballs).
- Not understood abstraction (even though they may have used it in the challenge.)

Examples of each are given below. Clarification of ambiguous language is provided using square brackets.

Understanding and Applying Abstraction Outside of Pirate Plunder

Generally, high-scoring participants (using Dr. Scratch, multiple-choice Scratch abstraction test, CTt and Pirate Plunder performance) were better able to apply abstraction to theoretical examples in Scratch. For example, one participant said that they could use custom blocks and cloning when creating a bowling game. Even going as far as to question whether cloning would be appropriate or not due to the way it works:

> **Researcher:** “Can you give me another example of where you’d use a custom block in Scratch?”
> **Participant:** “Take Pirate Plunder, say you needed to go to a certain place... In fact, you could create a bowling game and you could input the amount of power the ball would move, so you could determine how far it would go, or you could use it for some sort of game where you’d throw

or catapult something. So, you could change at different moments how far it would go."

Researcher: "Why have you used cloning?"

Participant: "Because if you do it then you use less sprites, you'd only have one sprite. And it also means that you can make all the sprites do the same thing because you can have 'when I start as a clone' and you can make it do the same thing."

Researcher: "In what other situation could you use cloning in Scratch?"

Participant: "Going back to the bowling game, I could probably clone the pins. But then if you cloned the bowling pins, some of them would (need to) go down and some of them wouldn't." [Here the participant has identified that the cloning may not be suitable for bowling pins because they will have different behaviour.]

In another example, a participant stated that they could use a custom block called 'dance' to make the Scratch cat perform a dance:

Researcher: "Can you give me another example of where you'd use a custom block in a different Scratch project?"

Participant: "If something was repeating over and over again and you wanted to lose less blocks then you could just do it in there. So if you wanted your Scratch to do a dance over and over again continuously then you could put it in there and just use 'dance', 'dance', 'dance'."

Understanding and Applying Abstraction Within Pirate Plunder

Other participants, who generally scored from mid-to-high on the assessment tasks, could explain the rationale for abstraction but only using examples from the Scratch challenge or Pirate Plunder. For example, one participant had produced an ideal custom block solution (similar to Figure 9.9 but without cloning), but could not explain where they could use abstraction outside of the Scratch challenge and Pirate Plunder context:

Researcher: "Why have you decided to use the 'moveAndTurn' custom block?"

Participant: "It's easier. I knew that there weren't any obstacles or anything and its quite easy for using lots of big numbers."

Researcher: "If there had of been obstacles, then what would you have done differently?"

Participant: "If there had of been obstacles then I would've added it on to here (the custom block), if not then I would've broken the blocks apart."

Researcher: "Can you give me another example of where you'd use a custom block in another project?"

Participant: “If you were doing a maze.”
Researcher: “Could you use one that’s not for moving or turning?”
Participant: “Yeah, you can do one for direction I think.”
Researcher: “Could you use one for anything else? In what situation would you think ‘oh I can use a custom block here’?”
Participant: “If it was lots of repeated moves, or if the turning was different each time.”

Limited Understanding of Abstraction

Lower-scoring participants tended to display a limited understanding of abstraction, even if they had used it in their post-Pirate Plunder Scratch challenge project. For example, this participant had used multiple custom blocks for different distances (lacking inputs for variable distance) and had almost got the solution to work:

Researcher: “Why have you used custom blocks?”
Participant: “I feel like custom blocks were really helpful during Pirate Plunder and I felt that they were also really helpful using Scratch. And because everyone was doing cloning (in Pirate Plunder) and I never did cloning I thought maybe if I use custom blocks, and I’m quite confident in that, then I can use them.”
Researcher: “Can you give me another example of where you’d use a custom block in Scratch?”
Participant: “I would probably use it... say so can you move them to this place, or move them to the second lamppost, you can probably do it in just five blocks doing a custom block but at the same time just put a repeat but I put a custom (block) cos it shows kind of that you know what you’re doing.”
Researcher: “Could you use custom blocks for something that’s not a move?”
Participant: “With a turn there’s not really any point in doing it. You could probably use it for a hide and show block. Like if you wanted to avoid the ghost ship in Pirate Plunder.”

Another participant had used one custom block for moving 40 steps, but had not fully understood how abstraction can be used to reduce duplication in the rest of the project:

Researcher: “Can you give me another example of where you’d use a custom block in another project?”
Participant: “I don’t know.”
Researcher: “In what situation would you think I could use a custom block there?”

Participant: "When it's like, 34 and 10, you can use a custom block because that's quite a lot, say two blocks."

9.4.6 Confidence Ratings

Table 9.11 shows the results of the Pirate Plunder post-test questionnaire. As with Study 3, the confidence ratings are on a scale from 3 to 0 (very confident to not confident) and confidence changes from 1 to -1 (improved to declined). Totals have been included in the bottom row. The custom block confidence ratings correlated significantly with the use of the correct custom block in the post-Pirate Plunder projects ($r(83) = .62$, $p < .001$) and cloning confidence ratings correlated significantly with correct use of cloning ($r(83) = .36$, $p = .001$).

Table 9.11: Descriptive statistics of the questionnaire confidence ratings for each group (PP/SP = Pirate Plunder/Spreadsheets, SP/PP = Spreadsheets/Pirate Plunder and SC/PP = Scratch/Pirate Plunder)

Condition		Scratch Confidence	Change	Custom Block Confidence	Change	Clone Confidence	Change
	M	1.54	1.00	1.96	0.64	1.43	0.25
PP/SP	*N*	28	28	28	28	28	28
	SD	1.14	1.85	1.11	0.56	1.17	0.80
	M	1.65	0.65	1.92	0.65	1.31	0.42
SP/PP	*N*	26	26	26	26	26	26
	SD	1.02	0.56	1.02	0.49	1.01	0.64
	M	1.95	0.90	1.97	0.90	1.36	0.64
SC/PP	*N*	29	29	29	29	29	29
	SD	0.83	0.31	0.98	0.31	0.95	0.63
	M	1.72	0.86	1.95	0.73	1.37	0.39
Total	*N*	83	83	83	83	83	83
	SD	1.00	1.13	1.02	0.47	1.04	0.70

9.4.7 CTt as a Predictor of Pirate Plunder Performance

As in Study 3, the CTt pre-test scores correlated significantly with both measures of Pirate Plunder performance: number of challenges completed, $r(88) = .53$, $p < .001$ (Figure 9.15) and total stars collected, $r(88) = .53$, $p < .001$ (Figure 9.16). It is worth noting that ceiling effects on the Pirate Plunder performance measures may have resulted in a weaker correlation than if the game did not have a maximum level.

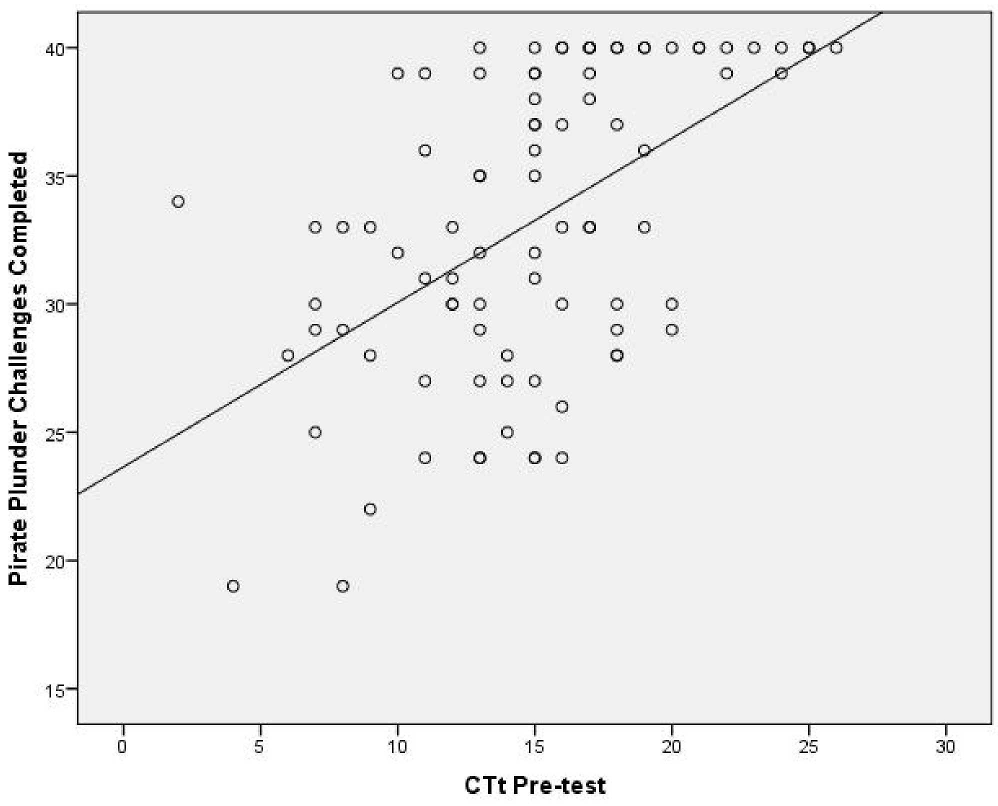


Figure 9.15: Relationship between CTt pre-test score and number of Pirate Plunder challenges completed for each participant (with regression line)

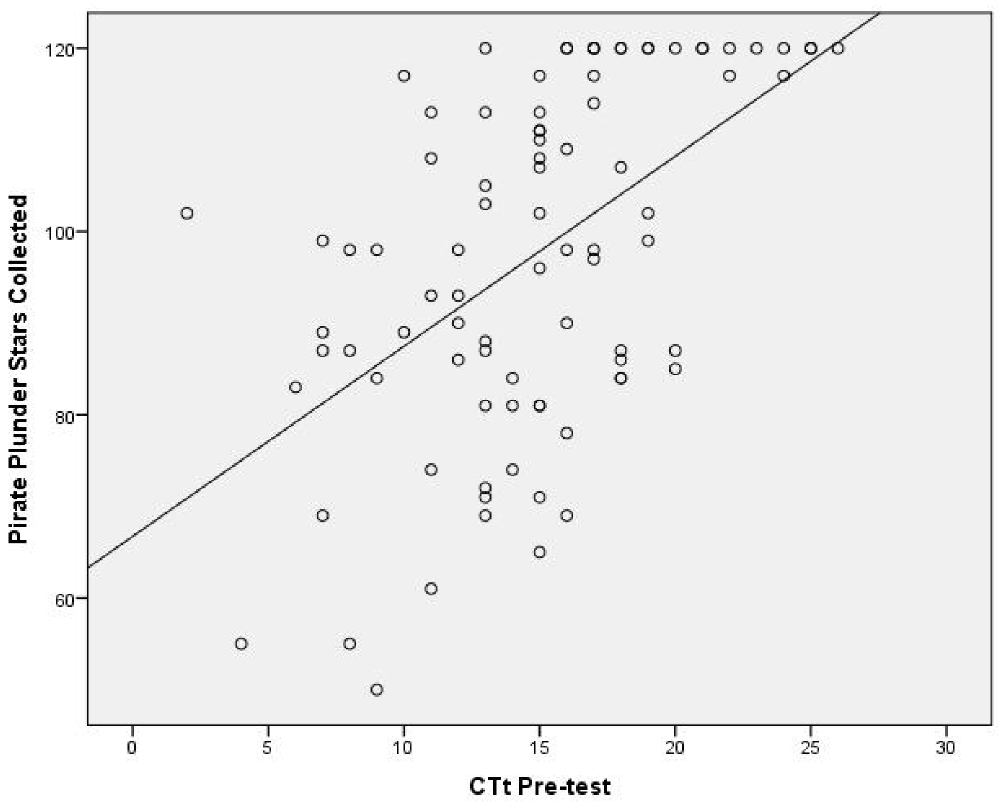


Figure 9.16: Relationship between CTt pre-test score and number of Pirate Plunder stars collected for each participant (with regression line)

9.5 Discussion

9.5.1 Was Pirate Plunder Effective?

Hypothesis 1 - Did Pirate Plunder Improve Abstraction in Scratch?

The results of both the Scratch challenge and the multiple-choice Scratch abstraction test after Phase 1 indicate that Pirate Plunder was effective in teaching primary-school children to use abstraction in Scratch, in comparison to a non-programming and a programming curriculum. Participants in the intervention group were able to use custom blocks and cloning to reduce duplication in a Scratch project.

However, the results from Phase 2 (after the crossover) are not as clear, in part due to the lack of a genuine control group. The Phase 2 intervention participants improved their scores on the Scratch challenge and were successful in using abstraction after playing Pirate Plunder. Yet, they did not significantly improve their scores on the multiple-choice Scratch abstraction test.

The Phase 1 Dr. Scratch results from the Scratch challenge show that 15/25 (60%) of the intervention group used custom blocks and a further 3/25 (12%) used cloning (note that these participants also used custom blocks). This is compared to one use of custom blocks and one use of cloning for the non-programming control and no use of either in the programming control. Whilst these results show that participants used abstraction in their Scratch projects, they do not show whether they used it correctly (to reduce duplication). These concerns come from the analysis by Robles et al. (2017) who found that duplication is still common in Scratch projects that use abstraction. The completeness criteria (Section 9.4.4) can be used to clarify this. Those results show that of those 18 to use custom blocks at mid-test, 10 used them correctly in line with the specification and none of them used cloning correctly to reduce sprite duplication. The single-use of both custom blocks and cloning in the spreadsheets group was also incorrect. At post-test, 29/55 (52.7%) participants in the Phase 2 intervention groups used custom blocks and a further 14 (25.4%) used cloning. When combined with the completeness criteria, 24 of those produced the correct custom block and 9 used cloning correctly.

The completeness criteria show that Pirate Plunder was effective in getting children to use abstraction correctly but less so than the Dr. Scratch results indicate. This is a weakness of using Dr. Scratch as a measure of CT, as discussed earlier in the chapter. The completeness criteria show that almost half (34/80) of participants were able to use custom blocks to reduce duplication after playing Pirate Plunder and 9 of these were able to use cloning to reduce the number of sprites in the project. A further third of participants (26/80) attempted to use custom blocks but did so incorrectly. This suggests that formative, project-based CT measures like Dr. Scratch need to be combined with software engineering metrics to assess whether functionality that indicates CT (or programming skills) has been used correctly.

The artifact-based interviews showed that similar to Study 3, the majority of interviewed participants had understood why they should use abstraction in Scratch after playing Pirate Plunder. Yet, as stated in Section 9.4.5, some participants could only apply this knowledge within the context of Pirate Plunder or the Scratch challenge assessment. This implies that whilst Pirate Plunder can be used to teach primary school children to use abstraction, this knowledge is only a starting point. By combining the game with formal teaching, children should be able to learn to apply these skills in other programming projects. This is discussed further in the next section and in the following chapter (Chapter 10).

The questionnaires showed that participants were more confident using Scratch, custom blocks and cloning after playing Pirate Plunder. This implies that they are more willing to use them in Scratch after playing the game and was demonstrated by a correlation between the custom block and clone confidence ratings and the use of both of those in the post-Pirate Plunder Scratch challenge.

Hypothesis 2 - Did Pirate Plunder Improve Computational Thinking?

The results of the Computational Thinking test show that Pirate Plunder improved CT compared to the non-programming curriculum after Phase 1 of the study. However, these results were not repeated after the crossover. Interestingly, the Phase 2 control group declined significantly on the CTt from mid-to post-test. This is likely because they were doing the same assessment for the third time during the study and had lost motivation to complete it properly.

9.5.2 Why Was Pirate Plunder Effective?

As discussed in the previous section, the majority of participants used abstraction in Scratch after playing Pirate Plunder. This section explores the reasons behind this, considering the changes made from Study 3.

Updated Assessments

The main reason for the efficacy of Pirate Plunder in comparison with Study 3 is that the updated Scratch challenge assessment was able to demonstrate improvements in using abstraction. It successfully addressed the two weaknesses of the Scratch assessment task discussed in Section 8.5.2: it was able to motivate participants to use abstraction and to restrict them from using the same strategy at pre-and post-test. The multiple-choice Scratch abstraction test then added a further measure of their ability to use abstraction in Scratch, asking them to apply their knowledge about custom blocks and cloning to specific questions.

By instructing participants to reduce the block count in an existing project, the Scratch challenge linked what participants had learnt in Pirate Plunder with the task itself. This enabled them to debug the duplicated code and refactor it without having to produce duplication first, as in the Scratch task from Study 3. This is the main reason why more participants used abstraction in the Scratch challenge despite the same time limit (40 minutes). Having the starting blocks (that reset the cat sprite) off to the right-hand side of the project and clear instructions not to delete them resulted in fewer participants removing these blocks. This meant that unlike Study 3, participants did not struggle with the cat sprite not resetting to its starting position after the program had finished. Furthermore, having clear rules on the task sheet and screenshots of the starting blocks on the reverse gave participants a clearer understanding of the task.

Procedure

The procedure used in Study 4 had participants playing Pirate Plunder twice a week. The shorter time between each session will have enabled the participants to better understand the learning content because they can retain more knowledge between each session. The intervention was also an average of 30 minutes longer in total than in Study 3.

Pirate Plunder Changes from Study 3

The changes to Pirate Plunder from Study 3 (Section 9.1.2) had a positive impact on participants' ability to use abstraction in the Scratch challenge, namely using the 'get property of' block and understanding Scratch coordinates. The help feature was also widely-used (an average of 73 times per player).

As previously stated, 9 participants successfully used cloning to reduce the number of lamppost sprites. This involved using the 'get property of' block to move the cloned lamppost to the position of the cat sprite. This is a significant improvement from Study 3, where no participants used the block. This shows that the 'get property of' tutorial was more effective in getting participants to be able to recall it and understand it.

Participants were also more comfortable using the coordinate position indicator in Scratch because the axes were removed from the Pirate Plunder grid. Participants had to use the Pirate Plunder coordinate indicator throughout the game to get the position of items.

Interestingly, despite the Pirate Plunder changes, better abstraction assessment results and the additional 30 minutes playing the game, participants performed similarly on Pirate Plunder in comparison with Study 3 (Table 9.12). One reason for this that the participants were younger (by half a year on average), so their similar performance reflects improvements to the game. Furthermore, based on the artifact-based interviews, the Study 3 intervention participants did understand abstraction and would have been able to use it had they been given the Study 4 assessments. The game changes, therefore, made it easier for younger children to understand abstraction and to reach a similar point as the older participants in Study 4.

Table 9.12: Descriptive statistics of Pirate Plunder performance for Studies 3 and 4

Study		Challenges Completed	Total Stars Collected	Average Stars per Level
	M	32.63	96.12	2.94
3	*N*	57	57	57
	SD	5.45	17.64	0.13
	M	33.13	97.44	2.93
4	*N*	90	90	90
	SD	5.84	18.79	0.14

In terms of the game analytics, Table 9.13 shows the Pirate Plunder analytics for the Study 3 and 4 versions of the game. The only significant difference is in the average time per level ($t(145) = 2.77$, $p = .006$, $d = 0.49$), which is to be expected because participants had more time playing the game but completed a similar number of levels.

Table 9.13: Pirate Plunder average attempts and time spent overall and per level for Studies 3 and 4

Study		Total Level Attempts	Average Attempts per Level	Total Level Time (HH:MM:SS)	Average Time per Level (Seconds)
	M	244.70	7.58	03:01:53	334.57
3	*N*	57	57	57	57
	SD	107.11	3.25	00:43:50	69.34
	M	260.77	8.16	03:25:56	380.89
4	*N*	90	90	90	90
	SD	124.40	4.16	00:57:15	113.54

Computational Thinking

Pirate Plunder and the Scratch curriculum were effective in improving scores on the CTt because they both involve CT. However, using the research and observations in Chapters 3 and 4, another reason for the improvement is that the CTt uses visual programming blocks similar to Scratch and Pirate Plunder. Several participants in the interviews pointed out the similarities between the CTt questions and Pirate Plunder programming.

As we have seen in earlier chapters, it is difficult to separate CT from computer science. The correlation between CTt pre-test scores and Pirate Plunder performance could be due to visual programming ability or CT ability, or a combination of both. This once again highlights the issues with CT definitions and assessments, as well as using it for the justification of teaching computer science in primary education. Furthermore, the issue with Dr. Scratch being unable to detect whether blocks have been used correctly means that whilst it gives a broad overview of CT use in Scratch projects, it is far from an ideal assessment. The next chapter (Chapter 10) discusses the implications of this.

9.5.3 Limitations

One limitation is that participants did the same CT test three times during the 10-week study. Whilst they were told each time that it was important to try their best, doing the same 45-minute assessment for the third time may have contributed towards the similar results (or decline in the case of the control) from mid-to post-test (Phase 2). However, the lack of valid and reliable CT assessments (that clearly measure the same construct) means that it would be difficult to select another test to compare against. Furthermore, the partial-crossover design that resulted in the repeated assessment meant that all participants were able to play Pirate Plunder.

9.5.4 Future Suggestions and Improvements

This section gives reasons and suggestions for future improvements to Pirate Plunder and the study design. These include the low use of cloning in the post-Pirate Plunder Scratch challenge, poor block and input naming, combining Pirate Plunder with formal lessons and CT assessments.

Cloning

Cloning in the post-Pirate Plunder Scratch challenge assessment was attempted by 17/80 (21.3%) participants and used successfully by 9 of these. This meant that many projects still contained duplicated lamppost sprites. This can be partially explained by the number of participants that did not reach the cloning challenges (37.8%) on Pirate Plunder, implying that the earlier levels took too long for the player to progress through during the given time. This is one of the difficulties with creating a game to teach concepts that players have no previous experience with, particularly ones that are cognitively challenging. It also highlights the broad range of abilities in a single school year group. However, the number of participants that used custom blocks in their post-Pirate Plunder Scratch challenge solutions (76.3%) implies the earlier custom block and inputs levels were effective. It may just have been that cloning required more time than was available during Studies 3 and 4. Even so, 62.2% of participants reached cloning levels but only 19.8% attempted to use them. One reason for this is that the transfer gap between creating cannonballs and creating lampposts was too big, as cannonballs in Pirate Plunder are created using a slightly different set of blocks.

This presents an opportunity for data mining of the game analytics to identify specific points where the game could be improved. This is discussed further in Chapter 10.

Poor Block and Input Naming

It was common in both Pirate Plunder and the Scratch challenge for participants to give their custom blocks and inputs 'bad' names. These were often random series of letters or words that did not correspond to what the block or input was doing (an example is shown in Figure 9.17). This is one of the Scratch code smells identified by Techapalokul (2017) and could be addressed in Pirate Plunder by validating block names to what the block does, with these restrictions teaching the player how to name blocks and variables correctly in Scratch.

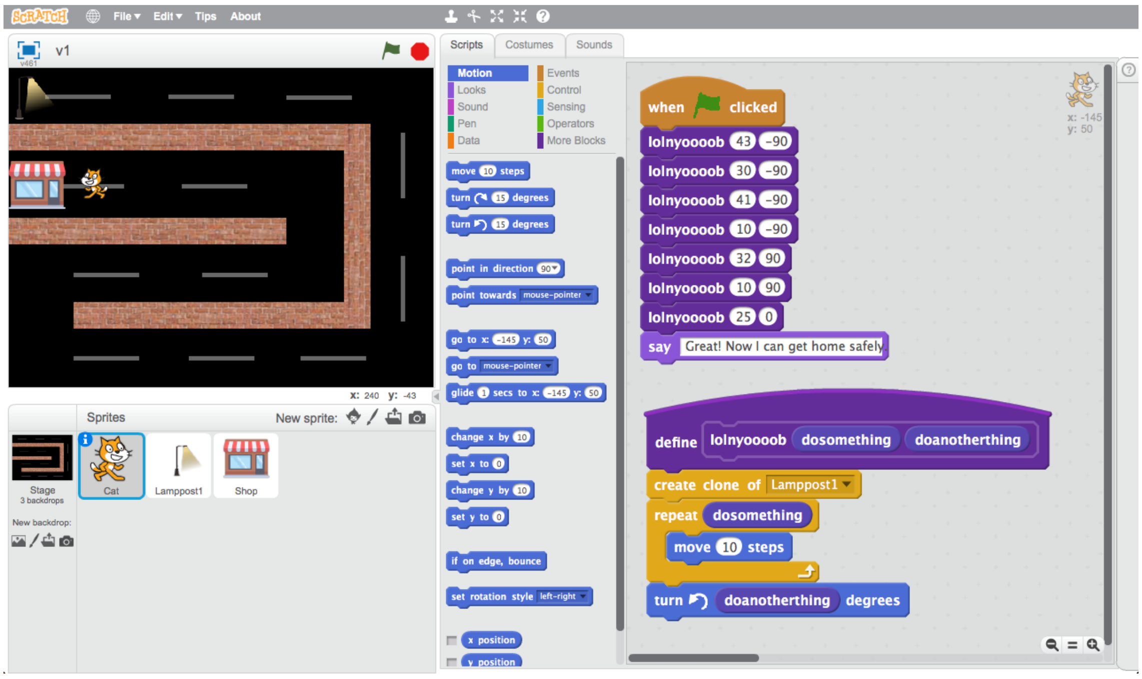

```
when [flag] clicked
lolnyooooob 43 -90
lolnyooooob 30 -90
lolnyooooob 41 -90
lolnyooooob 10 -90
lolnyooooob 32 90
lolnyooooob 10 90
lolnyooooob 25 0
say Great! Now I can get home safely.

define lolnyooooob dosomething doanotherthing
create clone of Lamppost1
repeat dosomething
    move 10 steps
turn ↺ doanotherthing degrees
```

Figure 9.17: Screenshot showing an example of bad custom block and input naming in an otherwise ideal Scratch challenge solution

Combining Pirate Plunder with Formal Lessons

Pirate Plunder could be combined with formal lessons to A) give children a better understanding of Scratch before they play the game and B) to explain the extract method and abstraction in more detail with contextual examples. In this study, the group that did the Scratch curriculum before playing Pirate Plunder performed better on the game compared to the other groups, completing more challenges and collecting more stars. This suggests that having some previous Scratch experience aids learning. Furthermore, the decline of CTt scores in the Phase 2 assessments of the Phase 1 intervention group highlights the need for the learner to continue using these techniques in Scratch projects.

These lessons would introduce Scratch programming in a more structured approach, focusing on producing ‘good’ Scratch code and avoiding the code smells listed in Section 5.2.3. The Pirate Plunder levels could then be used as an example of code reuse and using the extract method to refactor duplicated Scratch code. The potential of this is demonstrated by the improvement on the multiple-choice Scratch abstraction test for the group who did Scratch before playing Pirate Plunder.

Computational Thinking Assessments

This study has shown that the CTt is a predictor of Pirate Plunder progress. Yet, it is difficult to know whether this is because both contain visual programming, or both contain CT. This shows that it is difficult to create reliable measures of CT and suggests that existing CT assessments are better suited to measuring programming skills and should be used in primary education as part of a wider assessment strategy.

9.6 Conclusions

In conclusion, Pirate Plunder can be used to teach primary school children (age 10 and 11) to use abstraction in Scratch. Children were able to use custom blocks (in most cases) and cloning (in some cases) to reduce block and sprite duplication in Scratch through procedural abstraction using the extract method and code reuse. This supports the results of Gibson (2012) in demonstrating that primary school children can understand abstract computer science concepts if the learning content is presented in a structured way.

The artifact-based interviews suggest that higher-scoring children did develop a general understanding of abstraction. Yet, most children could only explain what they had learnt within the context of Pirate Plunder and the Scratch challenge. The study results demonstrate that it is possible to teach children to use abstraction (even if they cannot apply it more generally) using game-based learning and a structured level progression. The CTt results also show that Pirate Plunder improves CT, in line with the current literature.

The success of Pirate Plunder indicates that it could be used as part of school curricula, either alongside traditional teaching or as a standalone application. The difficulty progression worked well in introducing abstraction in a way that rationalised its use. The game could be extended to include more computer science concepts, such as conditionals and variables. It could be used to show how block-based code compares to text-based code, similar to Code.org.

The next chapter (Chapter 10) discusses the implications of these results and those of the rest of the thesis on computer science in primary education and CT.

Chapter 10

General Discussion

This chapter concludes the thesis by discussing the theoretical and practical implications of the results of this programme of work. The aims of the thesis outlined in Chapter 1 were to identify areas of weakness in computer science (CS) education (Chapters 2 to 5) and to design, create and evaluate a programming game to address these weaknesses (Chapters 6 to 9). The outcome was Pirate Plunder, a novel educational block-based programming game that can be used in primary schools to teach children how to use abstraction in Scratch.

The chapter describes the contributions of the thesis, before exploring the implications of these on computational thinking (CT), methodological approaches, game design, debugging-first learning approaches, abstraction and Scratch. It then highlights the limitations of the thesis, explores implications for the wider context of CS education and discusses future direction.

10.1 Contributions

10.1.1 Pirate Plunder

The novel educational block-based programming game, Pirate Plunder, is the main contribution of the programme of work. Study 4 showed that it is an effective method of teaching the conceptually difficult programming concept of abstraction to primary school children and can be used to support Scratch teaching and learning.

The game is unique in its learning outcomes for the target age group (abstraction for primary school children), difficulty progression, use of game design elements such as rewards and avatars and the level of polish for a research game. Additionally, Pirate Plunder's game mechanics, difficulty progression, engagement strategies and analytics all contribute towards its success. The tutorials and in-game feedback allow players to develop an understanding of the learning content without external assistance. The difficulty progression successfully scaffolds abstraction in a way that rationalises and justifies its use and enables children to transfer those skills in Scratch.

This is done by forcing the player to use abstraction in situations where it is beneficial. The engagement strategies of avatars and the reward system keep players motivated. The use of game analytics to adjust the level progression of Pirate Plunder show the importance of using data to improve game designs, particularly for conceptually difficult content.

10.1.2 Teaching Abstraction to Primary School Children

Abstraction is an essential skill in CS and the main tenet of current CT definitions, yet it is conceptually difficult for novices to understand (Kallia, 2017) and is, therefore, rarely taught in primary education. Despite these suggestions, Study 2 showed that primary school children (age 10 and 11) could recognise the benefits of using abstraction when asked to manipulate Scratch projects that used it. Study 4 then showed that children in this age group can learn to use abstraction in Scratch. After playing Pirate Plunder, children were able to use custom blocks and cloning to reduce block and sprite duplication in an existing Scratch project compared to two active control groups. Most of these children were then able to explain the benefits of abstraction when interviewed. Some children were also able to explain how they would use abstraction in other situations in Scratch.

10.2 Implications

10.2.1 Computational Thinking

What Do We Mean by Computational Thinking?

The difference in the definitions, models and frameworks discussed in Chapter 3 highlights the importance of clarifying what CT is. There is a growing consensus that CT is a universal problem-solving skill that will benefit every child (Grover et al., 2018) and an all-encompassing conceptual foundation that includes engineering, mathematics and design thinking (Shute et al., 2017). In theory, this argument works if all problem-solving, engineering, mathematics and design is done in computational domains. But it is not, and due to the lack of evidence of transfer of CT to non-computational domains (Denning et al., 2017), claims of it as a universal skill are unfounded.

The working definition of CT from Section 3.2.4: “CT is the thought processes involved in modelling and solving computational problems” supports the view of Armoni (2016) and Nardelli (2019) in suggesting that CT demonstrates the benefits of CS, providing a useful explanation of the skills required to be a good computer scientist. It is problematic to view CT as a separate discipline from CS, in terms of being used to solve ‘non-computational’ problems, as suggested by the results of Study 1 (Chapter 4). Just as mathematical thinking is not removed from mathematics. Furthermore, Nardelli's (2019) conclusion that CT should reach an "externally specified (set of) goals" (p. 34) is one of importance. He argues that as computer scientists differ from mathematicians in that they work towards computing an answer and not towards an equation defining the answer. Therefore, CT definitions should include working towards a set of pre-defined goals.

The results of the studies in this thesis suggest that the learning benefits of CT specific tasks are difficult to justify. Particularly learning materials like those of Barefoot (2019), for example, the abstraction task where children are asked to explain an animal without mentioning its name, requiring them to abstract the details of the animal. The focus instead should be on designing computational problems that require each CT skill, for example, abstraction or flow control, in a range of programming languages and tools. This then allows the learner to demonstrate that they have understood the concept and can use it to solve CS problems. Grover & Pea (2017) remind CT researchers that transfer of learning across contexts does not happen automatically and transfer of CT concepts to other learning contexts must be mediated. In which case, they become general problem-solving strategies or situated within other contexts and may lose their 'computational' nature.

How Do We Measure Computational Thinking?

The CT measures used in Studies 3 and 4, Dr. Scratch and the Computational Thinking test, both focus on visual programming and would be difficult to use to demonstrate improvements on non-computational tasks. Román-González et al. (2017) suggest that these two measures be combined with Bebras to create a complementary CT assessment. Bebras is a skill-transfer measure, yet despite being comprehension based, the questions are still largely computational (Figure 3.1 shows an example question). Román-González et al. state that the psychometric properties of Bebras are "still far from being demonstrated and some of them are at risk of being too tangential to the core of CT" (p. 158).

The working definition of CT as the 'thought processes' involved in modelling and solving computational problems implies that CT is a problem-solving process and is therefore difficult to accurately measure. In which case, assessments should focus on different types of programming problems that require CT skills. As suggested in Chapter 3, identifying the psychological constructs that underpin CT will help to clarify what it means, but not necessarily make it easier to measure and assess.

10.2.2 Methodological Approaches

Robust and quantitative experimental designs, like those used in this programme of work, are important because of the weaknesses with evaluating games for computing education. In a systematic literature review of educational games research, Petri &

von Wangenheim (2017) found that most evaluations used simple research designs, subjective feedback via questionnaires and small sample sizes. This is particularly important in the context of the thesis because programming games, amongst other tools, are being more widely used in primary education to teach programming, often without adequate classroom support.

Active Control Groups

The results of this programme of work support the need for active control groups in clarifying that the intervention condition is responsible for improvements in cognitive tasks (Simons et al., 2016). Using passive control groups means that the intervention and control groups are treated differently. This introduces psychological phenomena that can have an impact on results, such as the Hawthorne effect (Roethlisberger & Dickson, 1939), where the experimental group may change their behaviour because they are being observed. In the case of CS and CT research, these phenomena include excitement with using new technologies or approaches and interaction with researchers who have expertise in the field. Active control groups must be doing comparable tasks that expose them to a similar level of these new experiences. This has ethical implications for studies involving children in that all participants play an important role in the results and can do something different from their normal classroom experience. Whereas in studies with passive control groups, children can be disappointed when they realise they are not actively participating in the study.

Crossover Designs

Study 4 used a partial-crossover design that allowed all the study participants to play Pirate Plunder. This had important ethical implications because, during phase 1 of the study, the children in the control groups were asking when they would be able to play the game, despite actively participating in the study (following the spreadsheets or Scratch curriculum). This highlights that children can still be aware that there is something ‘more’ exciting that they were not doing, suggesting that game-based trials using children, in particular, should allow all participants to play the game as part of the study or getting to use new technology.

10.2.3 Designing Programming Games for Children

The success of Pirate Plunder has implications for designing programming games for children. These include learning content, player motivation, data mining and educational support.

Learning Content

Pirate Plunder was successful in introducing abstraction to primary school children in a way that rationalised and explained its use, as well as introducing basic Scratch movement and loops. This indicates that level design and difficulty progression are particularly important when designing educational programming games. The Pirate Plunder difficulty progression was effective because it enabled the player to become proficient in using blocks at each stage, before introducing new functionality, in this case, a method of code reuse that would then enable them to produce better solutions using fewer blocks. The levels were designed so that by the time a player was introduced to a code reuse method, they had been duplicating those blocks beforehand and could understand the benefit of using abstraction in that situation.

In Study 4, players were able to transfer the skills learnt in Pirate Plunder to Scratch. This was because the game used a similar layout and functionality to Scratch, including the block-based language, buttons and scene/program positions on the screen. More importantly, the differences in the block pallet, coordinate system and sprite use between the game and Scratch did not noticeably inhibit the transfer. This is important for game design because Pirate Plunder can be used alongside Scratch in the classroom, without external guidance about how the programming concepts transfer between the two.

Player Motivation and Engagement

Overall, players enjoyed Pirate Plunder and were motivated enough to continue playing it throughout each study, with several players requesting to be able to play it after the study had finished. Players commented positively on the tutorials (that the game shows you how to use the blocks), the difficulty progression (that you get more blocks as you progress through the game), the design (colours, user interface and background) and the collectable coins and customisable avatars. The performance statistics for Studies 3 and 4 support this (Table 9.12). Players continued playing, progressing and buying shop items throughout both studies.

The success of the collectable items, reward system and player avatars in keeping players engaged in the game suggests that these 'meta' game elements are an important part of designing learning games for children, particularly when they can compare these with other players.

Analytics and Data Mining

The field of data mining and big data has grown in recent years (Sin & Muthu, 2016). The amount of data produced by learning applications provides opportunities for this data to be used to improve technology and personalise content for the learner. In terms of game design, analytics can give valuable insights into how the game is being played and can enable the developer to adjust the application to further support learning, as was done throughout Pirate Plunder development.

The game analytics collected during development and Study 3 were key in adjusting the difficulty progression. Statistics were analysed on a per-level basis to identify the levels that players struggled on. This was particularly important when introducing custom blocks and cloning because these are conceptually difficult to understand. During the Study 3 pilot, performance on these levels was poor and resulted in a reduction of the number of earlier levels (to get the player using abstraction sooner) and simplification of the early custom block, input and cloning levels.

Teacher Support

In primary education, teachers are often concerned with their lack of ability and confidence (Rich et al., 2019) and feel isolated because they have to train themselves and find appropriate resources (Yadav et al., 2016). Often, children are given programming tools with little or no guidance and in some cases, CS is left out of lesson plans altogether. Programming games such as Pirate Plunder can allow teachers to 1) use these tools effectively in the classroom without having to know the content in-depth, 2) to train themselves using these tools, allowing them to build confidence in CS and 3) be used to indicate CS ability.

Pirate Plunder was played with minimal teacher and researcher support in Studies 3 and 4. This was achieved through the tutorials (Section 7.4.5) that introduced blocks and functionality by explaining and demonstrating what they are used for. The tutorials were an effective method of introducing concepts and were key to the success of the game. In addition, the in-game feedback and help feature meant that the player was better able to figure out how to correct their program, meaning that the teacher could focus on supporting weaker learners.

10.2.4 Debugging-First

The rationale for using a debugging-first approach in programming tasks comes from the completion strategy (Van Merriënboer & De Croock, 1992), where novice programmers modify or extend complete or incomplete programs (Paas, 1992). This approach was used in the text-based programming game, Gidget (M. J. Lee & Ko, 2014), which showed promising results in getting novices to learn programming concepts.

The results of Study 3 showed that a restrictive debugging-first approach in Pirate Plunder was no more beneficial to players than a non-debugging approach. This suggests that in programming games, a well-tested and looser debugging-first approach is a better option when it comes to improving player progress or learning outcomes, such as the one used in the Study 4 version of Pirate Plunder. This may be because novices struggle to understand code snippets, particularly containing blocks that are inexperienced with, so forcing them to use these blocks negates the benefits of a debugging-first approach.

In addition, the Scratch challenge assessment required the player to debug and refactor the existing program. This is compared to the Scratch task assessment in Study 3, where the player had to first produce a program and then debug and refactor it, which is a lot of functionality to produce in the 40 minutes they were given. This highlights the importance of designing programming tasks for children that teach them to understand code snippets that they have not produced themselves.

10.2.5 Abstraction

The success of using Pirate Plunder to teach abstraction supports the suggestions of Gibson (2012) in demonstrating that primary school children can understand abstract CS concepts if learning content is structured effectively. Furthermore, they support neo-Piagetian theory, which suggests that people, regardless of their age, progress through increasingly abstract forms of reasoning as they gain expertise in a domain (Lister, 2011). This is in comparison with traditional views that children only begin to reason abstractly once they reach a certain age. Yet, the most important finding is that the 'threshold' concept (one that opens up a new way of thinking about something) of procedural abstraction, which is difficult even for high school and university students to understand (Kallia & Sentance, 2017), can be taught to and understood by primary school children.

Chapter 2 analysed the wide range of programming tools designed for primary school children. Part of this analysis included the target age of tools and how much abstraction they allow. Those that allow limited abstraction are aimed at children age 8 and above and tools allowing full abstraction (through procedures) aimed at children age 10 and above. This suggests that if children are going to be using these programming tools for anything other than creative design, they should have at least some understanding of abstraction. Particularly in block-based languages where code smells are common.

It is important to consider the current state of CS education when discussing the relevance of individual CS and CT concepts. In countries where CS is compulsory at primary level, it lacks the teacher expertise required to implement it successfully (e.g. The Royal Society, 2017). Skills like procedural abstraction are only necessary if the learner already has some basic programming knowledge and is running into problems (e.g. duplication code smells) that further understanding can help to avoid. It may be better to let learners make mistakes when programming, before introducing them to the tools and skills that they can then use to solve these problems (Ginat, 2003) (using a similar approach to Study 2). However, this process would still need to be supervised by a teacher with adequate CS knowledge.

10.2.6 Scratch

The prevalence of bad programming habits and code smells in Scratch projects (Chapter 5) highlights issues with using constructionist, self-directed block-based programming tools, such as Scratch, to teach CS to children.

Whilst Scratch is good for fostering creativity, the open-ended nature of it allows children to create solutions using bad programming practices and code smells, such as widespread copying and pasting (duplication), dead blocks and long scripts (Meerbaum-Salant et al., 2011; Techapalokul, 2017). This is important because Scratch and similar tools (Hopscotch, Tynker, etc.) are widespread in primary education, meaning that children are often using these tools without adequate support and can end up forming bad programming habits (Aivaloglou & Hermans, 2016).

It was observed during the Scratch lessons in Studies 3 and 4 that when children were left to their own devices, they would ignore using programming functionality and focus instead on selecting and adding sprites, drawing and adding audio. This highlights the importance of structured teaching alongside constructionist tools, either through lesson plans that explain functionality, or game-based approaches where the child is guided through using CS concepts. Games such as Pirate Plunder are beneficial because they restrict the learner from reverting to their previous method of doing something, as is the case with code reuse in Pirate Plunder.

10.3 Limitations

10.3.1 Situated Use of Abstraction

It could be argued that the children in Study 4 have only learnt to use abstraction in a specific context because the Scratch assessment and Pirate Plunder both similarly use custom blocks and cloning (for sprite movement). This was highlighted during the artifact-based interviews, where most participants could only apply their understanding of abstraction in the context of Pirate Plunder or the Scratch assessment. Yet, the aim of the study and of Pirate Plunder itself was to see if primary school children were able to use abstraction, which is a conceptually difficult skill. The success of Pirate Plunder shows that primary school children can be taught to use procedural abstraction and code reuse. Pirate Plunder could, therefore, be used as a starting point for teaching children how and why to use abstraction in other programming languages. Furthermore, there were indications that higher-scoring children could apply abstraction in a wider context. Several were able to explain how they would use procedures for repetitive tasks that required parameters, such as a procedure for bowling a bowling ball that takes direction and power as its arguments.

10.3.2 Broader Programming Skillset

Pirate Plunder was designed to teach abstraction to primary school children. However, despite abstraction being an important skill in CS, these skills must be developed as part of a broader programming skillset. Moving forward, Pirate Plunder could be expanded to include other CS concepts, such as variables and conditionals. It could be used alongside a structured curriculum that contextualises some of the key points in other block-based or text-based languages and expanded to include a discovery section in the form of a level editor. Whilst the studies in this thesis concentrated on primary-school children, the game could be used by novice programmers of any age. Additional features could include paired-programming (having two players work on the same solution, similar to Pyrus (Shi, Shah, Hedman, & O'Rourke, 2019)) or having players check and grade the solutions of other players, highlighting where the solution could be improved and developing code comprehension skills.

10.3.3 Difficulties Designing Programming Games for Children

The results of the studies in this programme of work have shown that there are difficulties with designing programming games for children because of the differing ability levels between children of the same age. This was an issue with Pirate Plunder, where some players did not reach the later 'cloning' levels in the study time because they found the earlier levels too difficult. One way to solve this problem is to adapt game content based on how the player is doing, concentrating on areas of weaknesses that have been identified automatically by the game using analytics (Mees, Jay, Habgood, & Howard-Jones, 2017).

10.4 Wider Context

10.4.1 Supporting Teachers

This programme of research took place in seven different primary schools in northern England. Observations of computing practices within these schools confirm that teachers need more support if teaching CS in primary education is to be successful. Despite each school having adequate technology (IT suites and sets of high-spec tablets), the teaching of the national computing curriculum was sparse. Most of the schools used Scratch as their main programming tool, with children doing at least some computing lessons in either Year 5 or Year 6 (age 9 to 11). Yet, the majority of teachers lack the confidence or expertise to deliver a full curriculum, instead allowing children to follow Scratch's inbuilt tutorials or to play mathematics games such as Times Table Rockstars. Large teacher surveys by Sentance & Csizmadia (2017), Rich et al. (2019) and Yadav et al. (2016) show that this situation is not uncommon.

To counteract this, governments and policymakers need to support programs that provide CS training programs and resources to primary teachers. This is already happening in the UK through the Barefoot Computing Project, which was created in 2014 by the British Computer Society in collaboration with the Department for Education and BT. Their 2016 report (BT and Ipsos MORI, 2016) found that teachers were growing in confidence with delivering the computing curriculum (with 81% of the 400 teachers interviewed saying that they were confident with it.) Initiatives like this are vital for improving CS education in primary schools, as teachers are then able to explain CS concepts and have some understanding of what children are doing.

10.4.2 Educational Programming Tools

In addition to training programs and resources, educational programming tools play an important role in the success of CS education. With tablets now widespread in UK primary schools, visual programming tools and games can be used by teachers to support CS. Yet, it is difficult for teachers to know what programming tools to use as they differ considerably in cost, complexity and learning approach. Particularly when there is evidence that some tools can result in bad programming habits that can carry over to text-based languages. It is not unreasonable to suggest that a combination of games, creative tools and physical devices would offer the best all-round approach to CS. This would involve the child learning CS through games and formal teaching, then applying these skills in creative tools and using physical devices.

10.4.3 Computational Thinking

The issues surrounding CT lead to questions on how much emphasis there should be on it in CS education. Chapter 2 showed that CS is often combined and integrated with existing subjects, such as mathematics, informatics and digital literacy (Table 2.1). Teachers are then being told not only to get children to think about CT in CS lessons but also in other subjects beyond computing (BT and Ipsos MORI, 2016).

Governments and policymakers must clarify the aims and objectives of CS in primary education. If the purpose is to simply expose children to CS, giving them a basic understanding and building confidence with technology, then creative programming tools like Scratch can be effective. However, it is reasonable to suggest that today's children will need a good understanding of technology when they enter the job market. This means that they will need to develop an understanding of well-designed systems, which requires knowledge of programming and design concepts that make sense to children within the domain of CS.

The ideas surrounding CT have played an important role in getting CS into primary education. Yet, without a valid and reliable set of measures, it is difficult to show the transfer of CT to non-computational tasks and therefore substantiate the claims of it as a universal skill important for every child. It is reasonable to suggest that CT is useful in CS education as a useful demonstration of the skills required to solve computational problems. But, curricula should still focus on programming, keeping the learning grounded in functional skills that can be reliably assessed.

One aspect of CT definitions that is vital to all learners is the 'soft' problem-solving skills of persistence when facing difficult problems, being able to handle ambiguous, complex and open-ended problems and asking questions about why and how things work. If using CT in the classroom develops these skills, then it is reasonable to argue that it will benefit the learner, even if its goal of using computational ideas to solve non-computational problems is misguided.

10.5 Future Direction

10.5.1 Abstraction in Other Programming Tools

The thesis has concentrated on code smells and abstraction in Scratch. However, Chapter 5 also indicated that code smells can exist in other educational programming tools, such as Kodu Game Lab and GameMaker Studio 2. Future research could involve building analysis tools to identify code smells in these programming environments. It could then answer questions about how abstraction is relevant in these tools. Can it be used to reduce code smells in a similar way to custom blocks and cloning in Scratch? and how do you introduce good programming practices in programming tools that use different programming approaches?

10.5.2 Text-Based Abstraction

It would be interesting to see whether the abstraction skills learnt by children in Studies 3 and 4 benefit them when they move to text-based programming languages in secondary school and beyond. Particularly, are they able to see how functions and procedures are similar to custom blocks? and are they able to recognise duplicated code smells in text-based languages? This would require a longitudinal study to measure the effect of the Pirate Plunder learning content over time.

Another experiment could be conducted with older children (age 11 to 14) to see if Pirate Plunder benefits text-based programming in a shorter time frame. Perhaps having them play Pirate Plunder before a Python curriculum, answering similar research questions to those above.

10.5.3 Improving Programming Tools

The difficulties in teaching good software engineering practices using constructionist programming tools, like Scratch, suggests that programming tools can be improved to better teach CS concepts. Future research would focus on designing programming tools that place restrictions on what the learner can do, introducing programming concepts as they progress before allowing them to experiment in an open environment (a guided-discovery approach). This research would aim to answer questions about whether these changes make a difference compared to existing programming tools. Think-aloud studies could be used to understand the learners thought processes when using these tools.

10.5.4 Other Programming Concepts

Despite the focus on abstraction in this thesis, Section 10.3 explained the importance of these skills as part of a broader programming skill-set. Future research could focus on designing similar games, or expanding Pirate Plunder, to teach other programming concepts such as variables and conditionals. Answering similar questions to those in this study: can primary school children understand and learn to use these in other programming tools?

10.5.5 Attitudes Towards Computer Science

Another interesting question based on the studies in this thesis is whether participants have improved their attitudes towards programming and CS, and in turn, will this impact whether they go onto to pursue qualifications in CS.

10.5.6 Computational Thinking Research

Although this thesis moved away from designing measures to assess CT, it would be an interesting direction for future research. This would begin by designing a series of experiments to firstly design a measure of CT (using different CT concepts). The measure could then be used to measure improvements in non-computational tasks after programming interventions. This could go some way to answering the questions about CT and its usefulness in other disciplines.

10.5.7 Pirate Plunder in the Classroom

Finally, the success of Pirate Plunder indicates that it would be a useful part of CS curricula in primary and secondary education. This would involve expanding the game to introduce other CS principles and designing a curriculum and learning materials to accompany the game. Future versions of the game could also include transitions or comparisons to text-based programming, a level designer, paired programming or a solution checking mechanic where players have to approve or suggest improvements to other players' programs.

References

Aho, A. V. (2012). Computation and Computational Thinking. *The Computer Journal*, *55*(7), 833–835. doi:10.1093/comjnl/bxs074

Aivaloglou, E., & Hermans, F. (2016). How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 53–61). doi:10.1145/2960310.2960325

Alfieri, L., Brooks, P. J., Aldrich, N. J., & Tenenbaum, H. R. (2013). Does Discovery-Based Instruction Enhance Learning? *Journal of Educational Psychology*, *103*(1), 1–18. doi:10.1037/a0021017

Allsop, Y. (2018). Assessing Computational Thinking Process using a Multiple Evaluation Approach. *International Journal of Child-Computer Interaction*, *19*(March 2019), 30–55. doi:10.1016/j.ijcci.2018.10.004

Ambrósio, A. P., Xavier, C., & Georges, F. (2014). Digital Ink for Cognitive Assessment of Computational Thinking. In *Proceedings of the 2014 IEEE Frontiers in Education Conference*. doi:10.1109/FIE.2014.7044237

Anderson, L. W., Krathwohl, D. R., Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., . . . Wittrock, M. (2009). *A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy*. Longman.

Armoni, M. (2012). Teaching CS in Kindergarten: How Early Can the Pipeline Begin? *ACM Inroads*, *3*(4), 18–19. doi:10.1145/2381083.2381091

Armoni, M. (2013). On Teaching Abstraction in Computer Science to Novices. *Journal of Computers in Mathematics and Science Teaching*, *32*(3), 265–284.

Armoni, M. (2016). Computer Science, Computational Thinking, Programming, Coding: The Anomalies of Transitivity In K–12 Computer Science Education. *ACM Inroads*, *7*(4), 24–27. doi:10.1145/3011071

Austin, E. J., Deary, I. J., Gibson, G. J., McGregor, M. J., & Dent, J. B. (1998). Individual Response Spread in Self-Report Scales: Personality Correlations and Consequences. *Personality and Individual Differences*, *24*(3), 421–438. doi:10.1016/S0191-8869(97)00175-X

Bailey, R., Wise, K., & Bolls, P. (2009). How Avatar Customizability Affects Children's Arousal and Subjective Presence During Junk Food–Sponsored Online Video

Games. *CyberPsychology & Behavior*, *12*(3), 277–283. doi:10.1089/cpb.2008.0292

Barefoot Computing. (2019). Barefoot Computing Resources. Retrieved February 6, 2019, from https://www.barefootcomputing.org/

Baron-Cohen, S., Leslie, A. M., & Frith, U. (1986). Mechanical, Behavioural and Intentional Understanding of Picture Stories in Autistic Children. *British Journal of Developmental Psychology*, *4*, 113–125. doi:10.1111/j.2044-835X.1986.tb01003.x

Barr, V., & Stephenson, C. (2011). Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community? *ACM Inroads*, *2*(1), 48–54. doi:10.1145/1929887.1929905

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). Recognizing Computational Thinking Patterns. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 245–250). doi:10.1145/1953163.1953241

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable Programming: Blocks and Beyond. *Communications of the ACM*, *60*(6), 72–80. doi:10.1145/3015455. arXiv: 1705.09413

Bauer, A., Butler, E., & Popović, Z. (2015). Approaches for Teaching Computational Thinking Strategies in an Educational Game: A Position Paper. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 121–123). doi:10.1109/BLOCKS.2015.7369019

Bauer, A., Butler, E., & Popović, Z. (2017). Dragon Architect: Open Design Problems for Guided Learning in a Creative Computational Thinking Sandbox Game. In *Proceedings of the 12th International Conference on the Foundations of Digital Games* (pp. 1–6). doi:10.1145/3102071.3102106

Bebras. (2018). Statistics. Retrieved from https://www.bebras.org/?q=statistics

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer Science Unplugged: School Students Doing Real Computing Without Computers. *New Zealand Journal of Applied Computing & Information Technology*, *13*(1), 20–29.

Benitti, F. B. V. (2012). Exploring the Educational Potential of Robotics in Schools: A Systematic Review. *Computers & Education*, *58*(3), 978–988. doi:10.1016/j.compedu.2011.10.006

Benton, L., Kalas, I., Saunders, P., Hoyles, C., & Noss, R. (2018). Beyond Jam Sandwiches and Cups of Tea: An Exploration of Primary Pupils' Algorithm-Evaluation Strategies. *Journal of Computer Assisted Learning*, *34*(5), 590–601. doi:10.1111/jcal.12266

Bers, M. U. (2010). The TangibleK Robotics Program: Applied Computational Thinking for Young Children. *Early Childhood Research & Practice*, *12*(2), 1–20.

Bers, M. U., Flannery, L. P., Kazakoff, E. R., & Sullivan, A. (2014). Computational Thinking and Tinkering: Exploration of an Early Childhood Robotics Curriculum. *Computers & Education*, *72*, 145–157. doi:10.1016/j.compedu.2013.10.020

Bienkowski, M., Snow, E., Rutstein, D., & Grover, S. (2015). *Assessment Design Patterns for Computational Thinking Practices in Secondary Computer Science*. SRI International.

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., & Engelhardt, K. (2016). *Developing Computational Thinking in Compulsory Education: Implications for Policy and Practice*. Joint Research Centre. doi:10.2791/792158

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., & Franklin, D. (2013). Hairball: Lint-inspired Static Analysis of Scratch Projects. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (pp. 215–220). doi:10.1145/2445196.2445265

Boylan, M., Demack, S., Wolstenholme, C., Reidy, J., & Reaney-Wood, S. (2018). *ScratchMaths: Evaluation Report and Executive Summary*. Education Endowment Foundation.

Boyle, E. A., Hainey, T., Connolly, T. M., Gray, G., Earp, J., Ott, M., . . . Pereira, J. (2016). An Update to the Systematic Literature Review of Empirical Evidence of the Impacts and Outcomes of Computer Games and Serious Games. *Computers & Education*, *94*, 178–192. doi:10.1016/j.compedu.2015.11.003

Brackmann, C. P., Román-González, M., Robles, G., Moreno-León, J., Casali, A., & Barone, D. (2017). Development of Computational Thinking Skills through Unplugged Activities in Primary School. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (pp. 65–72). doi:10.1145/3137065.3137069

Brennan, K., & Resnick, M. (2012). New Frameworks for Studying and Assessing the Development of Computational Thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association* (pp. 1–25).

Brown, N. C. C., Sentance, S., Crick, T., & Humphreys, S. (2014). Restart: The Resurgence of Computer Science in UK Schools. *ACM Transactions on Computing Education*, *14*(2), 1–22. doi:10.1145/2602484. arXiv: arXiv:1502.07526v1

BT and Ipsos MORI. (2016). *Tech Literacy: A New Cornerstone of Modern Primary School Education*.

Calao, L. A., Moreno-León, J., Correa, H. E., & Robles, G. (2015). Developing Mathematical Thinking with Scratch: An Experiment with 6th Grade Students. In *Design for Teaching and Learning in a Networked World* (pp. 17–27). doi:10.1007/978-3-319-24258-3_2

Calvert, S. L. (2015). Children and Digital Media. In *Handbook of child psychology and developmental science* (pp. 375–415). doi:10.1002/9781118963418.childpsy410

Clements, D. H., & Gullo, D. F. (1984). Effects of Computer Programming on Young Children's Cognition. *Journal of Educational Psychology*, *76*(6), 1051–1058. doi:10.1037/0022-0663.76.6.1051

Code.org. (2018). Code.org. Retrieved January 15, 2019, from https://code.org/

Colburn, T., & Shute, G. (2007). Abstraction in Computer Science. *Minds and Machines*, *17*(2), 169–184. doi:10.1007/s11023-007-9061-7

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin*, *35*(1), 191. doi:10.1145/792548.611966

Cragg, L., & Gilmore, C. (2014). Skills Underlying Mathematics: The Role of Executive Function in the Development of Mathematics Proficiency. *Trends in Neuroscience and Education*, *3*(2), 63–68. doi:10.1016/j.tine.2013.12.001

Da Cruz Alves, N., Gresse Von Wangenheim, C., & Hauck, J. C. (2019). Approaches to assess computational thinking competences based on code analysis in K-12 education: A systematic mapping study. *Informatics in Education*, *18*(1), 17–39. doi:10.15388/infedu.2019.02

Dagiene, V., & Futschek, G. (2008). Bebras International Contest on Informatics and Computer Literacy: Criteria for Good Tasks. In *Proceedings of the International Conference on Informatics in Secondary Schools - Evolution and Perspectives* (pp. 19–30). doi:10.1007/978-3-540-69924-8_2

Dagiene, V., & Stupuriene, G. (2016). Bebras - a Sustainable Community Building Model for the Concept Based Learning of Informatics and Computational Thinking. *Informatics in Education*, *15*(1), 25–44. doi:10.15388/infedu.2016.02

Dasgupta, S., Hale, W., Monroy-Hernández, A., & Hill, B. M. (2016). Remixing as a Pathway to Computational Thinking. In *Proceedings of the 19th ACM Conference on Computer - Supported Cooperative Work & Social Computing* (pp. 1438–1449). doi:10.1145/2818048.2819984

De Luca, C. R., Wood, S. J., Anderson, V., Buchanan, J.-A., Proffitt, T. M., Mahony, K., & Pantelis, C. (2003). Normative Data From the Cantab. I: Development of Executive Function Over the Lifespan. *Journal of Clinical and Experimental Neuropsychology*, *25*(2), 242–254. doi:10.1076/jcen.25.2.242.13639

Denning, P. J. (2009). Beyond Computational Thinking. *Communications of the ACM*, *52*(6), 28. doi:10.1145/1516046.1516054

Denning, P. J. (2017). Remaining Trouble Spots With Computational Thinking. *Communications of the ACM*, *60*(6), 33–39. doi:10.1145/2998438

Denning, P. J., Tedre, M., & Yongpradit, P. (2017). Misconceptions About Computer Science. *Communications of the ACM*, *60*(3), 31–33. doi:10.1145/3041047

Department for Education. (2013). *The National Curriculum in England*.

Dijkstra, E. W. (1972). The Humble Programmer. In *Acm turing award lectures*. doi:10.1145/1283920.1283927

Dimitrov, D. M., & Rumrill, P. D. (2003). Pretest-Posttest Designs and Measurement of Change. *Work*, *20*(2), 159–165.

Dorling, M., & White, D. (2015). Scratch: A Way to Logo and Python. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 191–196). doi:10.1145/2676723.2677256

Du, C. (2009). Empirical Study on College Students' Debugging Abilities in Computer Programming. In *Proceedings of the 1st International Conference on Information Science and Engineering* (pp. 3319–3322). doi:10.1109/ICISE.2009.550

Dugard, P., & Todman, J. (1995). Analysis of Pre-Test-Post-Test Control Group Designs in Educational Research. *Educational Psychology*, *15*(2), 181–198. doi:10.1080/0144341950150207

Duncan, C., Bell, T., & Tanimoto, S. (2014). Should Your 8-Year-Old Learn Coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 60–69). doi:10.1145/2670757.2670774

Durak, H. Y., & Saritepeci, M. (2018). Analysis of the Relation Between Computational Thinking Skills and Various Variables With the Structural Equation Model. *Computers & Education*, *116*, 191–202. doi:10.1016/j.compedu.2017.09.004

Eteokleous, N. (2019). Robotics and Programming Integration as Cognitive-Learning Tools. In *Advanced methodologies and technologies in artificial intelligence, computer simulation, and human-computer interaction* (pp. 6859–6871). IGI Global.

Falloon, G. (2016). An Analysis of Young Students' Thinking When Completing Basic Coding Tasks Using Scratch Jnr. On the iPad. *Journal of Computer Assisted Learning*, *32*(6), 576–593. doi:10.1111/jcal.12155

Fessakis, G., Gouli, E., & Mavroudi, E. (2013). Problem Solving by 5–6 Years Old Kindergarten Children in a Computer Programming Environment: A Case Study. *Computers & Education*, *63*, 87–97. doi:10.1016/j.compedu.2012.11.016

Feurzeig, W., & Papert, S. (2011). Programming-Languages as a Conceptual Framework for Teaching Mathematics. *Interactive Learning Environments*, *19*(5), 487–501. doi:10.1080/10494820903520040

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging From the Student Perspective. *IEEE Transactions on Education*, *53*(3), 390–396. doi:10.1109/TE.2009.2025266

Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). Designing ScratchJr: Support for Early Childhood Learning Through Computer Programming. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 1–10). doi:10.1145/2485760.2485785

Förster, E.-C., Förster, K.-T., & Loewe, T. (2018). Teaching Programming Skills in Primary School Mathematics Classes: An Evaluation using Game Programming. In *Proceedings of the 9th IEEE Global Engineering Education Conference* (pp. 1504–1513). doi:10.1109/EDUCON.2018.8363411

Forsythe, G. E. (1959). The Role of Numerical Analysis in an Undergraduate Program. *The American Mathematical Monthly*, *66*(8), 651–662. doi:10.1080/00029890.1959.11989384

Fowler, A. (2012). Enriching Student Learning Programming Through Using Kodu. In *Proceedings of the 3rd Annual Conference of Computing and Information Technology, Education and Research* (pp. 33–39). doi:10.5604/16420136.970913

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. (2016). Initialization in Scratch : Seeking Knowledge Transfer. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 217–222). doi:10.1145/2839509.2844569

Gibson, J. P. (2012). Teaching Graph Algorithms to Children of All Ages. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (pp. 34–39). doi:10.1145/2325296.2325308

Ginat, D. (2003). The Greedy Trap and Learning From Mistakes. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 11–15). doi:10.1145/792548.611920

Giordano, D., & Maiorana, F. (2014). Use of Cutting Edge Educational Tools for an Initial Programming Course. In *Proceedings of the 2014 IEEE Global Engineering Education Conference* (pp. 556–563). doi:10.1109/EDUCON.2014.6826147

Gouws, L., Bradshaw, K., & Wentworth, P. (2013). Computational Thinking in Educational Activities: An Evaluation of the Educational Game Light-Bot. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 10–15). doi:10.1145/2462476.2466518

Grover, S., Jackiw, N., Lundh, P., & Basu, S. (2018). Combining Non-Programming Activities With Programming for Introducing Foundational Computing Concepts. In *Proceedings of the International Society of the Learning Sciences* (pp. 925–928). doi:10.22318/cscl2018.925

Grover, S., & Pea, R. (2017). Computational thinking: A competency whose time has come. *Computer science education: Perspectives on teaching and learning in school*, (December), 19–39. Retrieved from http://hub.mspnet.org/index.cfm/33300

Grover, S., & Pea, R. D. (2013). Computational Thinking in K-12: A Review of the State of the Field. *Educational Researcher*, *42*(1), 38–43. doi:10.3102/0013189X12463051

Grover, S., Pea, R., & Cooper, S. (2015). Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students. *Computer Science Education*, *25*(2), 199–237. doi:10.1080/08993408.2015.1033142

Guzdial, M. (2008). Paving the Way for Computational Thinking. *Communications of the ACM*, *51*(8), 25–27. doi:10.1145/1378704.1378713

Guzdial, M. (2015). Learner-Centered Design of Computing Education: Research on Computing for Everyone. In *Synthesis lectures on human-centered informatics* (pp. 1–165).

Habgood, M. P. J., & Ainsworth, S. E. (2011). Motivating Children to Learn Effectively: Exploring the Value of Intrinsic Integration in Educational Games. *Journal of the Learning Sciences*, *20*(2), 169–206. doi:10.1080/10508406.2010.508029

Harms, K. J., Rowlett, N., & Kelleher, C. (2015). Enabling Independent Learning of Programming Concepts Through Programming Completion Puzzles. In *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 271–279). doi:10.1109/VLHCC.2015.7357226

Harvey, B., Garcia, D., Paley, J., & Segars, L. (2012). Snap! (Build Your Own Blocks). In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 662–662). SIGCSE '12. doi:10.1145/2157136.2157351

Heintz, F., Mannila, L., & Farnqvist, T. (2016). A Review of Models for Introducing Computational Thinking, Computer Science and Computing in K-12 Education. In *Proceedings of the 2016 IEEE Frontiers in Education Conference* (pp. 1–9). doi:10.1109/FIE.2016.7757410

Hermans, F., & Aivaloglou, E. (2016). Do Code Smells Hamper Novice Programming? A Controlled Experiment on Scratch Programs. In *Proceedings of the IEEE 24th International Conference on Program Comprehension* (pp. 1–10). doi:10.1109/icpc.2016.7503706

Hermans, F., & Aivaloglou, E. (2017). To Scratch or Not to Scratch?: A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (pp. 49–56). doi:10.1145/3137065.3137072

Hermans, F., Stolee, K. T., & Hoepelman, D. (2016). Smells in Block-Based Programming Languages. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 68–72). doi:10.1109/VLHCC.2016.7739666

Hubwieser, P., & Mühling, A. (2014). Playing PISA with Bebras. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 128–129). doi:10.1145/2670757.2670759

Hubwieser, P., & Mühling, A. (2015). Investigating the Psychometric Structure of Bebras Contest: Towards Mesuring Computational Thinking Skills. In *Proceedings of the 2015 International Conference on Learning and Teaching in Computing and Engineering* (pp. 62–69). doi:10.1109/LaTiCE.2015.19

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer*. Addison Welsey.

Kalelioğlu, F. (2015). A New Way of Teaching Programming Skills to K-12 Students: Code.org. *Computers in Human Behavior*, *52*, 200–210. doi:10.1016/j.chb.2015.05.047

Kalelioğlu, F., Gülbahar, Y., & Kukul, V. (2016). A Framework for Computational Thinking Based on a Systematic Research Review. *Baltic Journal of Modern Computing*, *4*(3), 583–596.

Kallia, M. (2017). *Assessment in Computer Science courses: A Literature Review*. Royal Society.

Kallia, M., & Sentance, S. (2017). Computing Teachers' Perspectives on Threshold Concepts. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (pp. 15–24). doi:10.1145/3137065.3137085

Kazakoff, E. R., & Bers, M. U. (2011). The Impact of Computer Programming on Sequencing Ability in Early Childhood. In *Proceedings of the American Educational Research Association Conference*. Retrieved from http://ase.tufts.edu/DevTech/publications/aera%20handout%20sequencing.pdf

Kazakoff, E. R., & Bers, M. U. (2012). Programming in a Robotics Context in the Kindergarten Classroom: The Impact on Sequencing Skills. *Journal of Educational Multimedia and Hypermedia*, *21*(4), 371–391.

Kazakoff, E. R., & Bers, M. U. (2014). Put Your Robot In, Put Your Robot out: Sequencing Through Programming Robots in Early Childhood. *Journal of Educational Computing Research*, *50*(4), 553–573. doi:10.2190/EC.50.4.f

Kazakoff, E. R., Sullivan, A., & Bers, M. U. (2013). The Effect of a Classroom-Based Intensive Robotics and Programming Workshop on Sequencing Ability in Early Childhood. *Early Childhood Education Journal*, *41*(4), 245–255. doi:10.1007/s10643-012-0554-5

Keuning, H., Heeren, B., & Jeuring, J. (2017). Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 110–115). doi:10.1145/3059009.3059061

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A Study of the Development of Programming Ability and Thinking Skills in High School Students. *Journal of Educational Computing Research*, *2*(4), 429–458. doi:10.2190/BKML-B1QV-KDN4-8ULH

Lawanto, K., Close, K., Ames, C., & Brasiel, S. (2017). Exploring Strengths and Weaknesses in Middle School Students' Computational Thinking in Scratch. In *Emerging research, practice, and policy on computational thinking* (pp. 307–326). doi:10.1007/978-3-319-52691-1_19

Lazonder, A. W., & Harmsen, R. (2016). Meta-Analysis of Inquiry-Based Learning: Effects of Guidance. *Review of Educational Research*, *86*(3), 681–718. doi:10.3102/0034654315627366

Lee, M. J., Bahmani, F., Kwan, I., Laferte, J., Charters, P., Horvath, A., ... Ko, A. J. (2014). Principles of a Debugging-First Puzzle Game for Computing Education. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 57–64). doi:10.1109/VLHCC.2014.6883023. arXiv: arXiv:1011.1669v3

Lee, M. J., & Ko, A. J. (2014). A Demonstration of Gidget, a Debugging Game for Computing Education. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 211–212). doi:10.1109/VLHCC.2014.6883060

Lee, T. Y., Mauriello, M. L., Ahn, J., & Bederson, B. B. (2014). CTArcade: Computational thinking with games in school age children. *International Journal of Child-Computer Interaction*, *2*(1), 26–33. doi:10.1016/j.ijcci.2014.06.003

Liao, Y.-K. C., & Bright, G. W. (1991). Effects of Computer Programming on Cognitive Outcomes: A Meta-Analysis. *Journal of Educational Computing Research*, *7*(3), 251–268. doi:10.2190/E53G-HH8K-AJRR-K69M

Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference* (pp. 9–18).

Liu, Z., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding Problem Solving Behavior of 6-8 Graders in a Debugging Game. *Computer Science Education*, *27*(1), 1–29. doi:10.1080/08993408.2017.1308651

Livingstone, I., & Hope, A. (2011). *Next Gen: Transforming the UK Into the World's Leading Talent Hub for the Video Games and Visual Effects Industries*. NESTA. London, England. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/22268224

Lodico, M. G., Ghatala, E. S., Levin, J. R., Pressley, M., & Bell, J. A. (1983). The Effects of Strategy-Monitoring Training on Children's Selection of Effective Memory Strategies. *Journal of Experimental Child Psychology*, *35*(2), 263–277. doi:10.1016/0022-0965(83)90083-8

Luciana, M., & Nelson, C. A. (1998). The Functional Emergence of Prefrontally-Guided Working Memory Systems in Four-To Eight-Year-Old Children. *Neuropsychologia*, *36*(3), 273–293. doi:10.1016/S0028-3932(97)00109-7

MacLaurin, M. B. (2011). The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 241–245). doi:10.1145/1926385.1926413

Malone, T. W., & Lepper, M. R. (1987). Making Learning Fun: A Taxonomy of Intrinsic Motivations for Learning. In *Aptitude, learning, and instruction* (pp. 223–253). doi:10.1016/S0037-6337(09)70509-1

Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2008). Programming by Choice: Urban Youth Learning Programming With Scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education* (pp. 367–371). doi:10.1145/1352135.1352260

Maloney, J., Resnick, M., & Rusk, N. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, *10*(4), 1–15. doi:10.1145/1868358.1868363.http. arXiv: -

Manches, A., & Plowman, L. (2015). Computing Education in Children's Early Years: A Call for Debate. *British Journal of Educational Technology*, *48*(1), 191–201. doi:10.1111/bjet.12355

Mannila, L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). Computational Thinking in K-9 Education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference* (pp. 1–29). doi:10.1145/2713609.2713610

Margolis, J., & Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing*. MIT Press.

Mayer, R. E. (2004). Should There Be a Three-Strikes Rule Against Pure Discovery Learning? The Case for Guided Methods of Instruction. *American Psychologist*, *59*(1), 14–19. doi:10.1037/0003-066X.59.1.14

McGrew, K. S. (2009). CHC Theory and the Human Cognitive Abilities Project: Standing on the Shoulders of the Giants of Psychometric Intelligence Research. *Intelligence*, *37*(1), 1–10. doi:10.1016/j.intell.2008.08.004

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (pp. 168–172). doi:10.1145/1999747.1999796

Mees, M., Jay, T., Habgood, J., & Howard-Jones, P. (2017). Researching Adaptivity for Individual Differences in Numeracy Games. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play* (pp. 247–253). doi:10.1145/3130859.3131315

Melero, J., Hernández-Leo, D., & Blat, J. (2011). Towards the Support of Scaffolding in Customizable Puzzle-Based Learning Games. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications* (pp. 254–257). doi:10.1109/ICCSA.2011.64

Meyer, J. H. F., & Land, R. (2003). Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising Within the Disciplines. In *Improving student learning – ten years on*. Edinburgh.

Miljanovic, M. A., & Bradbury, J. S. (2016). Robot ON!: A Serious Game for Improving Programming Comprehension. In *Proceedings of the 2016 IEEE/ACM 5th In-*

ternational Workshop on Games and Software Engineering (pp. 33–36). doi:10.1109/GAS.2016.014

Moreno-León, J., & Robles, G. (2014). Automatic Detection of Bad Programming Habits in Scratch: A Preliminary Study. In *Proceedings of the 2014 IEEE Frontiers in Education Conference* (pp. 1–4). doi:10.1109/FIE.2014.7044055

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects Jesús. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 132–133). doi:10.1145/2818314.2818338

Moreno-León, J., Román-González, M., & Robles, G. (2018). On Computational Thinking as a Universal Skill: A Review of the Latest Research on This Ability. In *Proceedings of the 2018 IEEE Global Engineering Education Conference*. doi:10.1109/EDUCON.2018.8363437

Mühling, A., Ruf, A., & Hubwieser, P. (2015). Design and First Results of a Psychometric Test for Measuring Basic Programming Abilities. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 2–10). doi:10.1145/2818314.2818320

Nardelli, E. (2019). Do We Really Need Computational Thinking? *Communications of the ACM*, *62*(2), 32–35. doi:10.1145/3231587

Ozcelik, E., Cagiltay, N. E., & Ozcelik, N. S. (2013). The Effect of Uncertainty on Learning in Game-Like Environments. *Computers & Education*, *67*, 12–20. doi:10.1016/j.compedu.2013.02.009

Paas, F. (1992). Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach. *Journal of Educational Psychology*, *84*(4), 429–434. doi:10.1037/0022-0663.84.4.429

Papadakis, S., Kalogiannakis, M., & Zaranis, N. (2016). Developing Fundamental Programming Concepts and Computational Thinking With ScratchJr in Preschool Education: A Case Study. *International Journal of Mobile Learning and Organisation*, *10*(3), 187–202. doi:10.1504/IJMLO.2016.077867

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc.

Paris, A. H., & Paris, S. G. (2003). Assessing Narrative Comprehension in Young Children. *Reading Research Quarterly*, *38*(1), 36–77. doi:10.1598/RRQ.38.1.3

Passey, D. (2017). Computer Science (CS) in the Compulsory Education Curriculum: Implications for Future Research. *Education and Information Technologies*, *22*(2), 421–443. doi:10.1007/s10639-016-9475-z

Pea, R. D., & Kurland, D. M. (1984). On the Cognitive Effects of Learning Computer Programming. *New Ideas in Psychology*, *2*(2), 137–168. doi:10.1016/0732-118X(84)90018-7

Pérez-Marín, D., Hijón-Neira, R., Bacelo, A., & Pizarro, C. (2018). Can Computational Thinking Be Improved by Using a Methodology Based on Metaphors and

Scratch to Teach Computer Programming to Children? *Computers in Human Behavior*, *In Press*, 1–10. doi:10.1016/j.chb.2018.12.027

Perrenet, J., Groote, J. F., & Kaasenbrood, E. (2005). Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, *37*(3), 64–68. doi:10.1145/1070000/1067467

Petri, G., & Gresse von Wangenheim, C. (2017). How Games for Computing Education Are Evaluated? A Systematic Literature Review. *Computers & Education*, *107*, 68–90. doi:10.1016/j.compedu.2017.01.004

Piaget, J. (1970). *Science of Education and the Psychology of the Child*. Orion.

Piaget, J. (2001). *Studies in Reflecting Abstraction*. doi:10.4324/9781315800509

Price, C. B., & Price-Mohr, R. M. (2018). An Evaluation of Primary School Children Coding Using a Text-Based Language (Java). *Computers in the Schools*, *35*(4), 284–301. doi:10.1080/07380569.2018.1531613

Radiant Games. (2016). Box Island.

Repenning, A., & Sumner, T. (1992). Using Agentsheets to Create a Voice Dialog Design Environment. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's* (pp. 1199–1207). doi:10.1145/130069.130149

Repenning, A. (2017). Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets. *Journal of Visual Languages and Sentient System*, *3*(1), 68–89. doi:10.18293/VLSS2017-010

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779. arXiv: -

Rich, P. J., Browning, S. F., Perkins, M., Shoop, T., Yoshikawa, E., Belikov, O. M., . . . Shoop, T. (2019). Coding in K-8: International Trends in Teaching Elementary/Primary Computing. *TechTrends*, *63*(3), 311–329. doi:10.1007/s11528-018-0295-4

Robertson, J. (2019). *The relationship between Executive Functions and Computational Thinking*. Unpublished manuscript.

Robles, G., Hauck, J. C. R., Moreno-León, J., Román-González, M., Nombela, R., & Gresse von Wangenheim, C. (2018). On Tools That Support the Development of Computational Thinking Skills: Some Thoughts and Future Vision. In *Proceedings of the International Conference on Computational Thinking Education 2018* (pp. 129–132).

Robles, G., Moreno-León, J., Aivaloglou, E., & Hermans, F. (2017). Software Clones in Scratch Projects: On the Presence of Copy-and-Paste in Computational Thinking Learning. In *Proceedings of the 2017 IEEE 11th International Workshop on Software Clones* (pp. 31–37). doi:10.1109/IWSC.2017.7880506

Roethlisberger, F. J., & Dickson, W. J. (1939). *Management and the Worker*. Cambridge: Harvard University Press.

Román-González, M. (2016). Computational Thinking Test: Design Guidelines and Content Validation. In *Proceedings of EDULEARN15 Conference* (pp. 2436–2444). doi:10.13140/RG.2.1.4203.4329

Román-González, M., Moreno-León, J., & Robles, G. (2017). Complementary Tools for Computational Thinking Assessment. In *Proceedings of the International Conference on Computational Thinking Education 2017* (pp. 154–159).

Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2016). Which Cognitive Abilities Underlie Computational Thinking? Criterion Validity of the Computational Thinking Test. *Computers in Human Behavior*, *72*, 678–691. doi:10.1016/j.chb.2016.08.047

Román-González, M., Pérez-González, J.-C., Moreno-León, J., & Robles, G. (2018a). Can Computational Talent Be Detected? Predictive Validity of the Computational Thinking Test. *International Journal of Child-Computer Interaction*, *18*, 47–58. doi:10.1016/j.ijcci.2018.06.004

Román-González, M., Pérez-González, J.-C., Moreno-León, J., & Robles, G. (2018b). Extending the Nomological Network of Computational Thinking With Non-cognitive Factors. *Computers in Human Behavior*, *80*, 441–459. doi:10.1016/j.chb.2017.09.030

Rose, S. P. (2016). Bricolage Programming and Problem Solving Ability in Young Children: An Exploratory Study. In *Proceedings of the 10th European Conference for Game Based Learning* (pp. 914–921).

Scherer, R. (2016). Learning From the Past–The Need for Empirical Evidence on the Transfer Effects of Computer Programming Skills. *Frontiers in Psychology*, *7*, 7–10. doi:10.3389/fpsyg.2016.01390

Scratch Team. (2019). Scratch statistics. Retrieved January 16, 2019, from https://scratch.mit.edu/statistics/

Seiter, L. (2015). Using SOLO to Classify the Programming Responses of Primary Grade Students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 540–545). doi:10.1145/2676723.2677244

Seiter, L., & Foreman, B. (2013). Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (pp. 59–66). doi:10.1145/2493394.2493403

Sentance, S., & Csizmadia, A. (2017). Computing in the Curriculum: Challenges and Strategies From a Teacher's Perspective. *Education and Information Technologies*, *22*(2), 469–495. doi:10.1007/s10639-016-9482-0

Shein, E. (2014). Should Everybody Learn to Code? *Communications of the ACM*, *57*(2), 16–18. doi:10.1145/2557447

Shi, J., Shah, A., Hedman, G., & O'Rourke, E. (2019). Pyrus: Designing A Collaborative Programming Game to Support Problem Solving Behaviors. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1–12). doi:10.1145/3290605.3300886

Shipstead, Z., Redick, T. S., & Engle, R. W. (2012). Is Working Memory Training Effective? *Psychological Bulletin*, *138*(4), 628–654. doi:10.1037/a0027473

Shrestha, N., Barik, T., & Parnin, C. (2018). It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge. In *Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 177–185). doi:10.1109/VLHCC.2018.8506508

Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying Computational Thinking. *Educational Research Review*, *22*, 142–158. doi:10.1016/j.edurev.2017.09.003

Simons, D. J., Boot, W. R., Charness, N., Gathercole, S. E., Chabris, C. F., Hambrick, D. Z., & Stine-Morrow, E. A. L. (2016). Do "Brain-Training" Programs Work? *Psychological Science in the Public Interest*, *17*(3), 103–186. doi:10.1177/1529100616661983

Sin, K., & Muthu, L. (2016). Application of Big Data in Education Data Mining and Learning Analytics – A Literature Review. *ICTACT Journal on Soft Computing*, *05*(04), 1035–1049. doi:10.21917/ijsc.2015.0145

Sovic, A., Jagust, T., & Sersic, D. (2014). How to Teach Basic University-Level Programming Concepts to First Graders? In *Proceedings of the 2014 IEEE Integrated STEM Education Conference* (pp. 1–6). doi:10.1109/ISECon.2014.6891050

Statter, D., & Armoni, M. (2016). Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 80–83). doi:10.1145/2978249.2978261

Stolee, K. T., & Fristoe, T. (2011). Expressing Computer Science Concepts Through Kodu Game Lab. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 99–104). doi:10.1145/1953163.1953197

Straw, S., Bamford, S., & Styles, B. (2017). *Randomised Controlled Trial of Code Clubs*. National Foundation for Educational Research and Raspberry Pi Foundation. Slough, England.

Strawhacker, A., Lee, M., & Bers, M. U. (2017). Teaching Tools, Teachers' Rules: Exploring the Impact of Teaching Styles on Young Children's Programming Knowledge in ScratchJr. *International Journal of Technology and Design Education*, *28*(2), 347–376. doi:10.1007/s10798-017-9400-9

Sullivan, A., Bers, M. U., & Mihm, C. (2017). Imagining, Playing, and Coding with KIBO: Using Robotics to Foster Computational Thinking in Young Children. In

Proceedings of the International Conference on Computational Thinking Education 2017 (pp. 110–115).

Swidan, A., Hermans, F., & Smit, M. (2018). Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (pp. 151–159). doi:10.1145/3230977.3230995

Techapalokul, P. (2017). Sniffing Through Millions of Blocks for Bad Smells. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 781–782). doi:10.1145/3017680.3022450

Techapalokul, P., & Tilevich, E. (2015). Programming Environments for Blocks Need First-Class Software Refactoring Support: A Position Paper. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 109–111). doi:10.1109/BLOCKS.2015.7369015

The Royal Society. (2012). *Shut Down or Restart? The Way Forward for Computing in UK Schools*.

The Royal Society. (2017). *After the Reboot: Computing Education in UK Schools*.

Thurstone, L. L. (1938). *Primary Mental Abilities*. Chicago: University of Chicago Press.

Tsarava, K., Moeller, K., & Ninaus, M. (2018). Training Computational Thinking Through Board Games: The Case of Crabs & Turtles. *International Journal of Serious Games*, *5*(2), 25–44. doi:10.17083/ijsg.v5i2.248

Turkle, S., & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, *11*(1), 3–33.

Twinkl Educational Publishing. (2018). Twinkl. Retrieved May 21, 2018, from https://twinkl.co.uk

U.S. Department of Labor. (2018). Bureau of Labor Statistics. Retrieved July 25, 2018, from https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Van Merriënboer, J. J. G., & Sweller, J. (2005). Cognitive Load Theory and Complex Learning: Recent Developments and Future Directions. *Educational Psychology Review*, *17*(2), 147–177. doi:10.1007/s10648-005-3951-0. arXiv: 0705.2802

Van Merriënboer, J. J. G., & De Croock, M. B. M. (1992). Strategies for Computer-Based Programming Instruction: Program Completion vs. Program Generation. *Journal of Educational Computing Research*, *8*(3), 365–394. doi:10.2190/MJDX-9PP4-KFMT-09PM

Vellerand, R. J., Gauvin, L. I., & Halliwell, W. R. (1986). Negative Effects of Competition on Children's Intrinsic Motivation. *The Journal of Social Psychology*, *126*(5), 649–657. doi:10.1080/00224545.1986.9713638

Webb, M., Davis, N., Bell, T., Katz, Y. J., Reynolds, N., Chambers, D. P., & Sysło, M. M. (2017). Computer Science in K-12 School Curricula of the 2lst Century:

Why, What and When? *Education and Information Technologies*, *22*(2), 445–468. doi:10.1007/s10639-016-9493-x

Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*. doi:10.1016/j.jtcvs.2015.01.023

Werner, L., Denner, J., & Campe, S. (2012). The Fairy Performance Assessment : Measuring Computational Thinking in Middle School. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 215–220). doi:10.1145/2157136.2157200

Wilson, A., & Moffat, D. C. (2010). Evaluating Scratch to Introduce Younger Schoolchildren to Programming. In *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*.

Wilson, C. (2015). Hour of Code – A Record Year for Computer Science. *ACM Inroads*, *6*(1), 22. doi:10.1145/2723168

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33. doi:10.1145/1118178.1118215

Wing, J. M. (2008). Computational Thinking and Thinking About Computing. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, *366*, 3717–3725. doi:10.1109/IPDPS.2008.4536091

Yadav, A., Gretter, S., Hambrusch, S., & Sands, P. (2016). Expanding Computer Science Education in Schools: Understanding Teacher Experiences and Challenges. *Computer Science Education*, *26*(4), 235–254. doi:10.1080/08993408.2016.1257418

Yadav, A., Stephenson, C., & Hong, H. (2017). Computational thinking for teacher education. *Communications of the ACM*, *60*(4), 55–62. doi:10.1145/2994591

Zelazo, P. D., Carter, A., Reznick, J. S., & Frye, D. (1997). Early Development of Executive Function: A Problem-Solving Framework. *Review of General Psychology*, *1*(2), 198–226. doi:10.1037/1089-2680.1.2.198

Zhong, B., Wang, Q., Chen, J., & Li, Y. (2016). An Exploration of Three-Dimensional Integrated Assessment for Computational Thinking. *Journal of Educational Computing Research*, *53*(4), 562–590. doi:10.1177/0735633115608444

Appendices

A Pirate Plunder Development

This section describes the software development of Pirate Plunder, as a supplement to Chapter 7. It explains the development stack and how the application was structured, specific details of the client and server projects, the local development pipeline, deployment and project management.

A.1 Stack

Pirate Plunder is a web application with the majority of the code written in TypeScript (a syntactical superset of JavaScript). It uses the MEAN stack: a MongoDB database, Express.js for routing requests, Angular for the front-end and Node.js as the platform. The application is split into a 'client' project that handles the front-end functionality of the game, and a 'server' API project that routes requests and handles the database. The projects run separately on different ports and communicate with each other using HTTP. Figure 1 shows a diagram of the system.

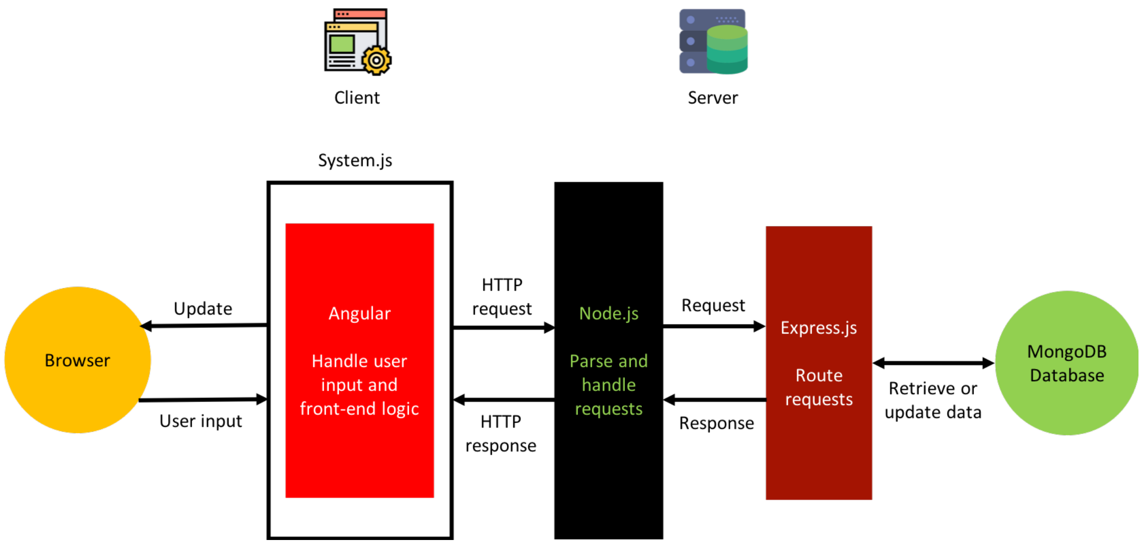


Figure 1: Diagram of the Pirate Plunder system architecture

Having a web-based game meant that data could be stored in a central database. This made player comparison easier and meant that the game did not have to be installed on school computers. In addition, once the study was finished, players could continue playing the game at home (if parental consent had been given).

A.2 Client

The client project is an Angular project written in TypeScript (that is then compiled down to JavaScript to be executed by the browser). The application is loaded into the browser through the 'index.html' file, which imports the system.js configuration file responsible for handling the execution of the Angular application. The client project uses the npm package manager to handle external dependencies.

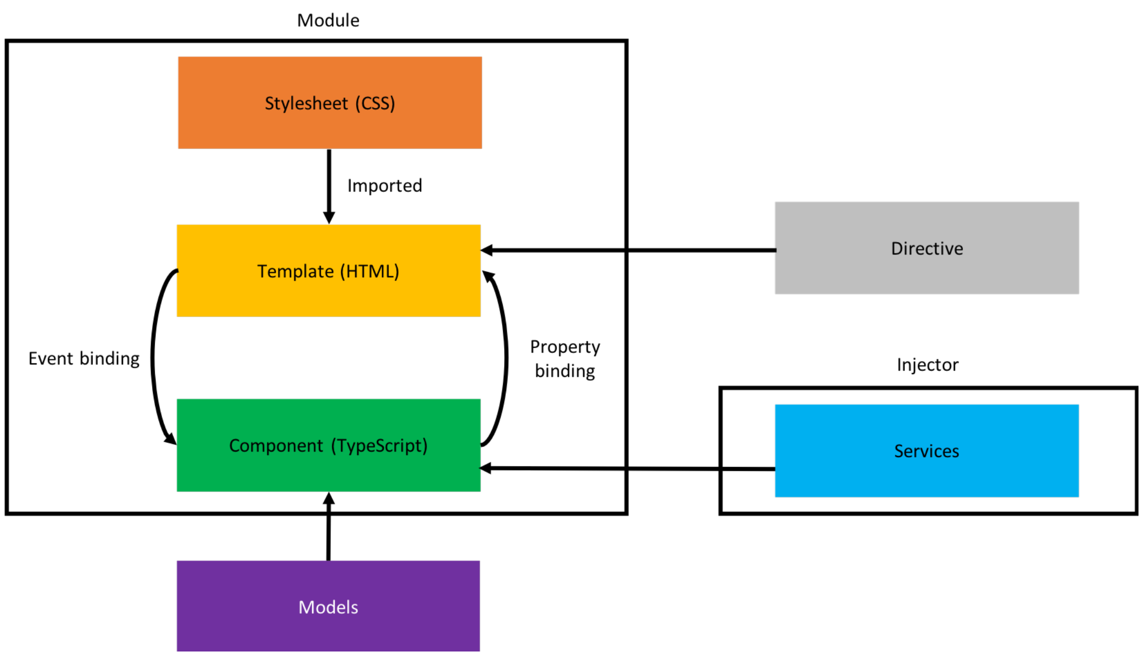


Figure 2: Diagram of typical Angular 2+ architecture

Structure

Angular uses a modular architecture with modules, components, services and directives as the main class types. Components and services can be grouped into modules. Pirate Plunder uses a single 'app' module. Components define 'views', which are screen elements that can be modified (in isolation) according to program logic and data. Components then use 'services' that can be injected into multiple components and specify reusable application logic. Models define TypeScript classes that represent different types of information. Figure 2 shows how these parts fit together.

The client application was structured as follows:

```
client/
├─ app/
│  ├─ about/
│  ├─ admin/
│  ├─ analytics/
│  ├─ blockly/
│  ├─ character/
│  ├─ character-select/
│  ├─ class/
│  ├─ game/
│  ├─ grid/
│  ├─ help/
│  ├─ info/
│  ├─ instructions/
│  ├─ level/
│  ├─ level-select/
│  ├─ login/
│  ├─ main-menu/
│  ├─ player-info/
│  ├─ shop/
│  ├─ star-rating/
│  └─ utils/
├─ assets/
└─ data/
```

Each folder in the ‘app’ folder (apart from ‘utils’) contains a component that has a TypeScript file for its logic, a HTML file for its template and a Less file that compiles down to CSS for the stylesheet. Services and models sit within the relevant folder. Less is an extension language for CSS that allows for variables and functions to be defined that can be reused in different files. There are components for each ‘page’ in the application (e.g. ‘about’, ‘admin’, ‘character-select’, ‘level’), then subcomponents that fit within these (e.g. ‘grid’, ‘blockly’). For example, the grid component is structured as follows, with ObjectIds specifying an enum that identifies each object in the grid:

```
grid/
├─ models/
│  ├─ CoinMesh.ts
│  ├─ GridPosition.ts
│  └─ Properties.ts
├─ grid.component.html
├─ grid.component.less
├─ grid.component.css
├─ grid.component.ts
├─ grid.service.ts
├─ mesh.service.ts
└─ ObjectIds.ts
```

The ‘assets’ folder contains the images and sounds used by the application. The ‘data’ folder contains JSON files that specify the levels (split into tutorials and challenges), shop items and configuration settings.

Frameworks

Several external frameworks are used within the client application: Google Blockly for the block-based programming environment, PixiJS for the game grid (previously Three.js for Study 3) and howler.js for handling audio. The Google Blockly framework was extended to produce a block-based language similar to Scratch. The framework does not come with Scratch type blocks so these had to be created manually. Google Blockly works by parsing the player’s block-based code into JavaScript, it was then executed in a service that handles each line of code, updating the grid accordingly.

Assets

The majority of the assets in Pirate Plunder were acquired from free sources and are used under the Creative Commons license. They are each attributed to the author on the ‘about’ screen. The avatar images are taken from a bought pack and some additional shop items have been created by the author of the thesis.

A.3 Server

The server is a Node.js application that runs separately to the client. It handles HTTP requests sent to the specified API endpoints, the Express.js framework then routes these requests to the Mongoose 'controllers' and 'models' that interact with the MongoDB database. The project uses JWT (JSON web tokens) to authenticate requests to certain endpoints.

Mongoose models specify the structure of the data because MongoDB is a NoSQL database, meaning that data is stored in documents and does not have to contain consistent fields. Controllers then handle the methods that can be called on that data. (e.g. adding a player to the database or logging in). The Mongoose controllers and models in Pirate Plunder are split into 'admin', 'analytics' and 'playerInfo'. Admin handles login from the administration section, analytics handles the game analytics and playerInfo handles the information about the player, including password hashing, current avatar, completed levels, etc.

The server has a configuration file for the 'origin' that can send requests to it and an environment file that contains the name of the current database. This means that different values can be used for local development and the live server.

A.4 Local Development

During local development, the client project is run using 'lite-server', a node server that serves the application, opens it in the browser and refreshes when HTML or JavaScript files are changed. This is used concurrently with 'tsc', which compiles the TypeScript files down to JavaScript when they are edited. The client also uses gulp.js, a toolkit that can be used to automate development tasks. In this case, it watches for changes to the Less files and compiles them down to CSS when they are edited. The server project is run using 'nodemon', a utility that uses Node.js and automatically restarts the process when files are changed.

GitFlow and Semantic Versioning

The projects were stored in separate Git repositories and were managed using the GitFlow branching model. GitFlow splits branches into master (major releases), develop, features, hotfixes and releases. This easily isolates new development from finished work. Both projects were versioned using the semantic versioning strategy (major, minor, patch), meaning that each change can be traced to a specific version of the game, making development and documentation easier.

A.5 Deployment

Pirate Plunder was hosted on a DigitalOcean 'droplet' (a remote server) running Ubuntu. It was deployed using continuous integration, which meant that changes could easily be pushed to the live version on the server from each Git repository. A Codeship service was set up to listen for new releases that had been pushed to the master branch. It would then run tests before updating the code on the server. Codeship was also configured to run a gulp task to 'bundle' the client project file into a single file, saving load time in the browser.

The server ran the client project using the 'http-server' module and the server project using Node.js.

A.6 Project Management

The project was managed using Trello, a free web-based project management application. This used the Kanban workflow methodology, with columns for 'Backlog', 'In Progress', 'In Review/Blocked', 'Bugs' and 'Done'. Each card was assigned a priority (low, medium or high). Figure 3 shows a burndown chart of the tasks completed throughout development.

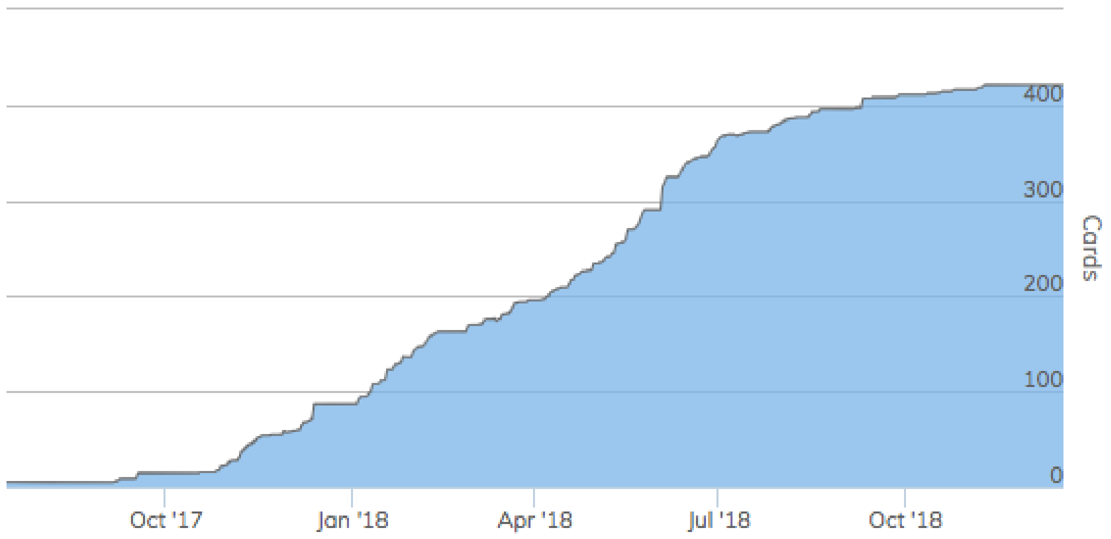


Figure 3: Chart of the use of Trello during the development of Pirate Plunder

B Other Development

B.1 Anonymising Projects

A C# application was developed to anonymise Scratch projects. The participants in Studies 3 and 4 would save projects in a folder of their name, using various file names. The application converted these folder names to participants' corresponding ID numbers and renamed each project as a version (V1, V2, etc.). This meant that completed projects are not identifiable without the paper sheet of ID numbers.

B.2 Collating Analytics

A Node.js application was developed to collate analytic data by player or by level. The MongoDB database stores analytics in a single collection with analytic differentiated using a 'type' field. The application connects to the database and uses the MongoDB driver to collate data and output it to a CSV file.

B.3 Adding Player Accounts

The player accounts application is a Node.js application that takes a list of names from a CSV and sends requests to the API so that they are added to the game database.

C Computational Thinking - Ethics Application

Sheffield Hallam University

APPLICATION FOR RESEARCH ETHICS APPROVAL (SHUREC2A)

SECTION A: Research Protocol

Important Note - If you have already written a research proposal (e.g. for a funder) that answers the methodology questions in this section please include a copy of the proposal and leave those questions blank. You **MUST** however complete **ALL** of Section B and C (risk assessment).

1. **Name of principal investigator:** Simon Rose

   Faculty: ACES

   Email address: simon.p.rose@student.shu.ac.uk

2. **Title of research:** Investigating the use of visual programming tools when developing computational thinking skills in young children

3. **Supervisor** (if applicable)**:** Dr. Jacob Habgood

   Email address: j.habgood@shu.ac.uk

4. **CONVERIS number (applicable for externally funded research):**

5. **Other investigators (within or outside SHU)**

Title	Name	Post	Division	Organisation

6. **Proposed duration of project**

   Start date: 1st May 2017 **End Date: 1st July 2018**

7. **Location of research if outside SHU:** Primary Schools in Sheffield

8. **Main purpose of research:**

   ☒ Educational qualification
   ☐ Publicly funded research
   ☐ Staff research project
   ☐ Other (Please supply details)

9. **Background to the study and scientific rationale** (500- 750 words approx.)

   Background

   Today's children will go on to live a life dominated by computing, both in the home and at work (Barr & Stephenson, 2011). Computing education is entering classrooms worldwide, with the aim of developing digital, media and information literacies. Several countries have introduced computer science into national curricula (Heintz, Mannila, & Farnqvist, 2016), meaning that children as young as 5-years-old are now taught basic programming. The need for children to be effective users of computational tools has led to the re-examination of the concept of ‘computational thinking’ (Wing, 2006), which has played a significant role in defining computer science learning content for children (Manches & Plowman, 2015).

Learning to program is seen as a way of developing computational thinking skills (Brennan & Resnick, 2012), which has led to the release of a variety of new software and robotics products designed for young children. Whilst there is evidence to suggest that children aged between 4 and 7 can basic programming skills (Bers, 2010; Fessakis, Gouli, & Mavroudi, 2013; Wohl, Porter, & Clinch, 2015), there is a comparative lack of empirical research into how programming tools are used by this age group.

Scientific Rationale

Visual programming tools, like Scratch (Resnick et al., 2009), are seen as a means of teaching programming skills to children (Berland & Wilensky, 2015; Wilson & Moffat, 2010). Most these tools use block-based programming, which is regarded as an effective method of teaching novice programming (Kelleher & Pausch, 2005; Mannila et al., 2014). This is because it lowers the barrier of entry for programming, whilst still introducing computer science concepts like conditionals and procedures.

However, there are some suggestions that block-based programming may encourage children to develop unusual programming approaches. Meerbaum-Salant, Armoni and Ben-Ari observed that 14 and 15-year-olds using Scratch took both top-down and bottom-up programming approaches to the extreme (2011). Furthermore, research from the author found that children aged 6 and 7 approached block-based programming differently depending on prior visual reasoning ability (Rose, 2016). This raises questions over the suitability of current visual programming environments to teach programming principles to young children.

The programme of research will involve several empirical studies investigating how these programming tools are used by young children. The first of these will study the effect of a programming game on the sequencing ability of children aged 5 and 6. Computer programming involves algorithmic thinking, in which objects or actions are put in sequence (Zelazo, Carter, Reznick, & Frye, 1997). Sequential thinking is a key concept in computing, language and mathematical curricula for this age group and is a fundamental part of cognitive development in children (Samara & Clements, 2004). Programming games designed for this age group, such as Lightbot, are often based on sequencing instructions, which suggests that they will help develop sequential thinking ability in young children.

These studies would then feed into the design, development and evaluation of a visual programming environment, which would be the primary contribution to knowledge of the PhD.

10. **Has the scientific / scholarly basis of this research been approved?** (For example by Research Degrees Subcommittee or an external funding body)

☒ Yes
☐ No - to be submitted
☐ Currently undergoing an approval process
☐ Irrelevant (e.g. there is no relevant committee governing this work)

11. **Main research questions**

We are seeking FREC approval for the first phase of PhD research, where we aim to investigate how children use existing programming tools. The research questions are therefore exploratory in nature:

- How do young children interact with visual programming tools?
- Does this interaction have any effect on the learning outcomes of these tools?
- Which visual programming tools are most effective in teaching programming principles?

The second phase of research will involve the building and evaluation of a visual programming environment. An ethics application for this will be submitted at a later date.

12. Summary of methods including proposed data analyses

The studies in the programme of research will follow a quasi-experimental pre-test post-test design and will be similar in terms of ethical procedure. The participants will be between 4 and 7 years old. However, school age groups may be used for individual studies, for example, Year 1 (5 and 6 years old) or Year 2 (6 and 7 years old).

Participants will complete a computer-based pre-test to gauge computational thinking skills, which may be broken down into individual skills such as sequencing. They will then be divided into two groups for an intervention, each using a different software application. The studies will examine how the children interact with the software, and whether it affects their overall computational thinking skills. The participants will then complete a post-test.

The primary measure of these studies will be the pre-and post-test scores. These will be analysed to test for differences between and within groups before and after the intervention. Quantitative data will also be collected as interaction data that is recorded by the software. Qualitative data will be researcher observations and participant feedback during the intervention.

A similar study was approved by FREC and carried out as part of a Masters in December 2015. The participants of that study were in Year 2 (6 and 7 years old).

SECTION B

1. Describe the arrangements for selecting/sampling and briefing potential participants. *This should include copies of any advertisements for volunteers, letters to individuals/organisations inviting participation and participant information sheets. The sample sizes with power calculations if appropriate should be included.*

The participants for the studies will be children aged between 4 and 7. Permission to perform each of the studies will be obtained via letters to the Headteachers of local primary schools (an example can be seen in Appendix 1).

For the first study, the sample size will be three classes of approximately 30 children (90 participants in total). Future studies will have a sample size of 60 participants or above (a minimum of two classes). They will be briefed by their teacher and again by the researchers before the study takes place (an example of the briefing given to a child by a researcher can be seen in Appendix 3).

2. What is the potential for participants to benefit from participation in the research?

Participants may benefit from the exposure to programming tools like those used in the school curriculum. This may also positively influence their computational thinking ability and problem-solving skills, and help them work towards the computing national curriculum for Key Stage 1. The participants may also get some enjoyment from using the tools.

3. **Describe any possible negative consequences of participation in the research along with the ways in which these consequences will be limited.**

   Participants could potentially misinterpret the study as a test of themselves rather than the software, so it will be made clear at the start of the study that this is not a test of the participants and doesn't count towards their school work in any way. They will be specifically informed that it is the software we are testing and reminded that they can drop out of the study at any time.

   It is expected that the studies will take place during school hours. So, there may be negative consequences in missing out on normal teaching time. To limit this, it will be ensured that the programming tools used are relevant to their current learning outcomes.

4. **Describe the arrangements for obtaining participants' consent.** *This should include copies of the information that they will receive & written consent forms where appropriate. If children or young people are to be participants in the study details of the arrangements for obtaining consent from parents or those acting in loco parentis or as advocates should be provided.*

   A parental consent form will be sent out by each school detailing the study taking place. The form in Appendix 2 represents the starting point for this consent form, but the exact contents will be adapted per the wishes of the school. Each child will be told (in child-friendly language) that they are taking part in a research project, it's nothing to do with their school work or assessments, and that data will be recorded but their names won't be. They will also be told that they don't have to take part and can stop at any time they like, and if they have any questions they can ask at any time (see Appendix 3 for the full script).

5. **Describe how participants will be made aware of their right to withdraw from the research.** *This should also include information about participants' right to withhold information and a reasonable time span for withdrawal should be specified*.

   Each child will be told in child-friendly language that they can withdraw from the study at any time. This option will be given to them when first participating in the study so they have sufficient time to decide whether they would like to participate or not.

6. **If your project requires that you work with vulnerable participants describe how you will implement safeguarding procedures during data collection.**

   Throughout the study, a member of school staff will be present in the room whenever possible. The school safeguarding procedures will be followed and any problems will be deferred to the teacher (such as behavioural issues).

7. **If Disclosure and Barring Service (DBS) checks are required, please supply details**

   Although this is not technically required, a DBS check has already been carried out by the school on the principle investigator. Certificate number: 001505136041

8. **Describe the arrangements for debriefing the participants.** This should include copies of the information that participants will receive where appropriate.

   Each child will be thanked for their contribution and asked if they have any feedback to give. It will be reaffirmed that the study has nothing to do with their school work or assessments and all their answers are anonymous.

9. **Describe the arrangements for ensuring participant confidentiality.** This should include details of:
   - how data will be stored to ensure compliance with data protection legislation

- how results will be presented
- exceptional circumstances where confidentiality may not be preserved
- how and when confidential data will be disposed of

Quantitative data will be stored with an assigned identification number. A paper copy of names and identification numbers will be stored separately to the data. Qualitative data will be collected as notes and written up with altered names were necessary (full detail of this can be seen in the data management plan in Appendix 4). The results will be presented in the form of a thesis and research papers and will be anonymous. There are no exceptional circumstances in which confidentiality will not be preserved. The list of names will be disposed of after data collection is complete.

10. **Are there any conflicts of interest in you undertaking this research?** (E.g. are you undertaking research on work colleagues or in an organisation where you are a consultant?) Please supply details of how this will be addressed.

None.

11. **What are the expected outcomes, impacts and benefits of the research?**

The expected outcome of the first study is that the programming game will positively sequencing ability, compared with a phonics game. This research will then feed into several more studies, the focus of which is dependent on the results.

Overall, it is hoped that the research will make a valuable contribution to the body of research on young children's computational thinking.

12. **Please give details of any plans for dissemination of the results of the research. This includes your plans for preserving and sharing your data. You may refer to your attached Data Management Plan.**

It is hoped the research will be published as part of a PhD thesis and as research papers. Data will be stored in the SHURDA, and made available to other institutions upon request (see Appendix 4 for more detail).

SECTION C

HEALTH AND SAFETY RISK ASSESSMENT FOR THE RESEARCHER

1. **Will the proposed data collection take place on campus?**

☐ Yes (Please answer questions 4, 6 and 7)
☒ No (Please complete all questions)

2. **Where will the data collection take place?**
(Tick as many as apply if data collection will take place in multiple venues)

	Location	**Please specify**
☐	Researcher's Residence	
☐	Participant's Residence	
☒	Education Establishment	Primary Schools in Sheffield (First study at Southey Green Community Primary)
☐	Other e.g. business/voluntary organisation, public venue	

☐ Outside UK

3. **How will you travel to and from the data collection venue?**

☐ On foot ☐ By car ☒ Public Transport
☐ Other (Please specify)

Please outline how you will ensure your personal safety when travelling to and from the data collection venue

Travel with a reputable public transport company.

4. **How will you ensure your own personal safety whilst at the research venue?**

Make myself aware of the fire procedure and any other necessary safety procedures in place. I will also sign in to the visitor's book so that I am on record as being in the building in case of emergency.

5. **If you are carrying out research off-campus, you must ensure that each time you go out to collect data you ensure that someone you trust knows where you are going (without breaching the confidentiality of your participants), how you are getting there (preferably including your travel route), when you expect to get back, and what to do should you not return at the specified time.** (See Lone Working Guidelines). Please outline here the procedure you propose using to do this.

I will inform my housemate of my whereabouts, travel route and the time I'm expecting to be back. If I do not return on time then he is to try and contact me initially, then the school and/or public transport company.

6. **Are there any potential risks to your health and wellbeing associated with either (a) the venue where the research will take place and/or (b) the research topic itself?**

☒ None that I am aware of
☐ Yes (Please outline below including steps taken to minimise risk)

7. **Does this research project require a health and safety risk analysis for the procedures to be used?**

☐ Yes
☒ No

(If **YES** the completed Health and Safety Project Safety Plan for Procedures should be attached)

Adherence to SHU policy and procedures

Personal statement	
I confirm that: • this research will conform to the principles outlined in the Sheffield Hallam University Research Ethics policy • this application is accurate to the best of my knowledge	
Principal Investigator	
Signature	Simon Rose
Date	17/02/2017

Supervisor (if applicable)	
Signature	J Habgood
Date	17/02/2017

Please ensure the following are included with this form if applicable, tick box to indicate:

	Yes	No	N/A
Research proposal if prepared previously	☐	☒	☐
Any recruitment materials (e.g. posters, letters, etc.)	☒	☐	☐
Participant information sheet	☒	☐	☐
Participant consent form	☒	☐	☐
Details of measures to be used (e.g. questionnaires, etc.)	☐	☒	☐
Outline interview schedule / focus group schedule	☐	☒	☐
Debriefing materials	☐	☒	☐
Health and Safety Project Safety Plan for Procedures	☐	☒	☐
Data Management Plan*	☒	☐	

If you have not already done so, please send a copy of your Data Management Plan to rdm@shu.ac.uk
It will be used to tailor support and make sure enough data storage will be available for your data.

D Computational Thinking - Data Management Plan

Exploring the effect of programming tools on young childrenâ€™s computational thinking

Project Name Exploring the effect of programming tools on young childrenâ€™s computational thinking

Principal Investigator / Researcher Simon Rose

Description Research conducted for a PhD

Institution Sheffield Hallam University

Data collection

What data will be produced?

Quantitative data in the form of pre-test and post-test answers from a computer-based application. This will be stored digitally in a MongoDB database in BSON format. MongoDB works well with the language the application will be written in (JavaScript), and is fast and easy to maintain. BSON is a lightweight file format (based on JSON) that does not take up much storage space (MB's). The raw data can be exported as JSON or CSV, or to SPSS for storage on SHURDA or the university research drive. MongoDB also handles file versioning. The data will be analysed using Node.js to extract overall scores for participants, which will be stored in an SPSS file.

Qualitative data in the form of written observations and participant feedback. This will be collected as notes and written up in rich text format (rtf).

Data documentation

How will your data be documented and described?

The quantitative data will be documented using metadata. It will be stored in a database that will have suitable column names that should be readable to someone who is not associated with the project. The SPSS file that the data will be analysed using will be documented in a readme text file, and have understandable column names. The observation notes will be written up and accompanied with a readme to describe ambiguous comments and structure.

Ethical and copyright issues

How will you deal with any ethical and copyright issues?

As the participants are children, consent will be obtained from their parents or carers. This consent form will clarify that parents are giving consent for the researchers to collect data, and that this data may be shared with other institutions.

All data will be anonymised and assigned an ID number that is user dependent. Names recorded in the observation notes will be changed before they are stored.

The copyright of the data will lie with the University, and it will be stored in SHURDA once collected.

Data storage

How will your data be structured, stored and backed up?

Data will be logged to a MongoDB database from a JavaScript web application for the pre-test and post-test. It will be sent from the application to a NoSQL database on a cloud platform (such as DigitalOcean). Backups of the data will be placed on the University research drive. Study observations will be written up as soon as possible and stored on the research drive until the study is complete. These observations will follow a filename format that includes the date and session in which they were recorded.

Data preservation

What are the plans for the long-term preservation of data supporting your research?

The quantitative data will be stored long-term, as it is anonymised and will not need to be deleted on any contractural grounds. It is hoped that the data will be used in a published research publication, and can be used to facilicate further analysis.

The data will be stored in the SHURDA upon completion of the study.

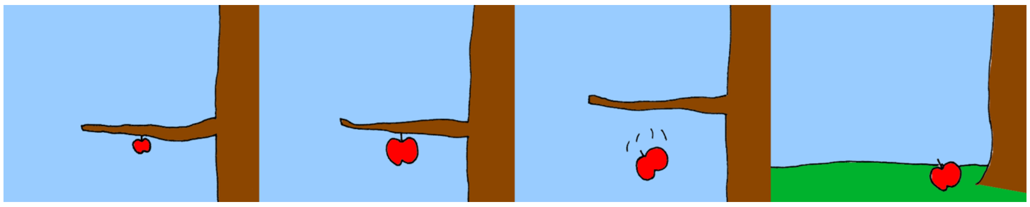
Data sharing

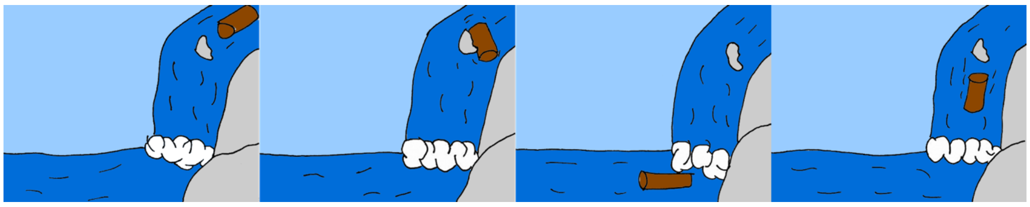
What are your plans for data sharing after submission of your thesis?

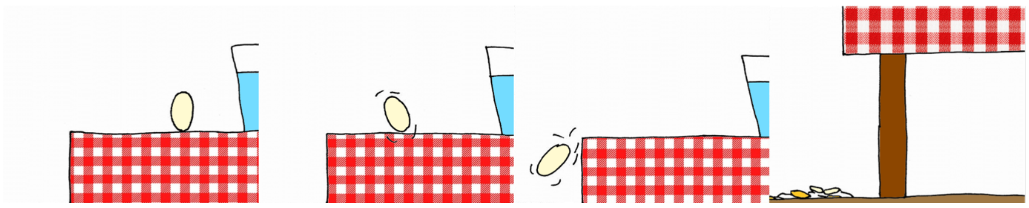
It is hoped that the data will be useful to other researchers and can be further analysed. The data will be stored in the SHURDA, and be given to other institutions upon request.

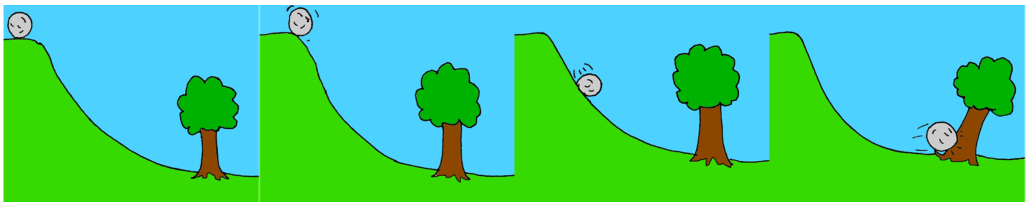
E Study 1 - Story Sequences

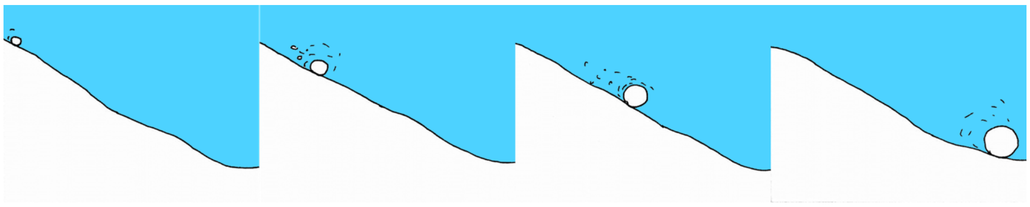
Mechanical 1 (objects interacting causally with each other)

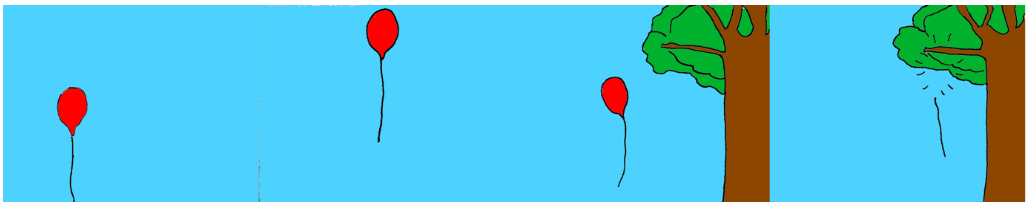




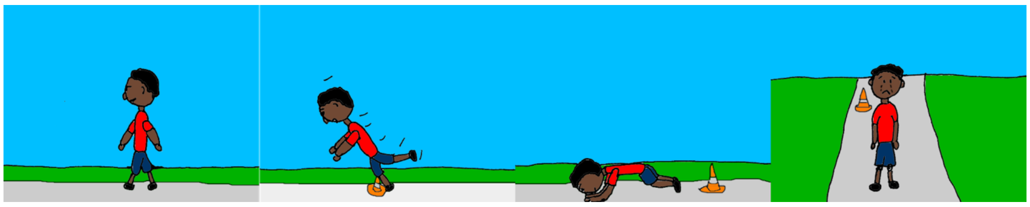


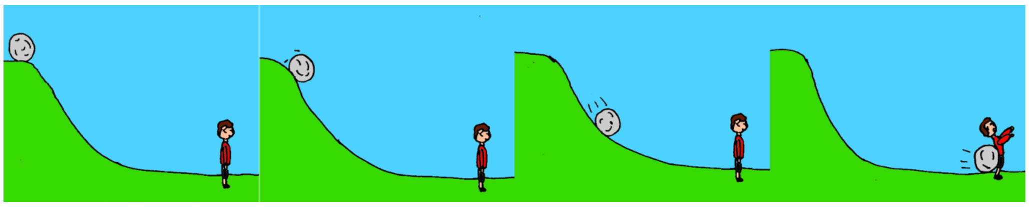


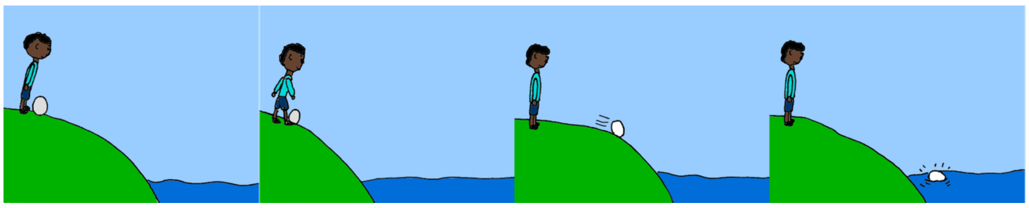


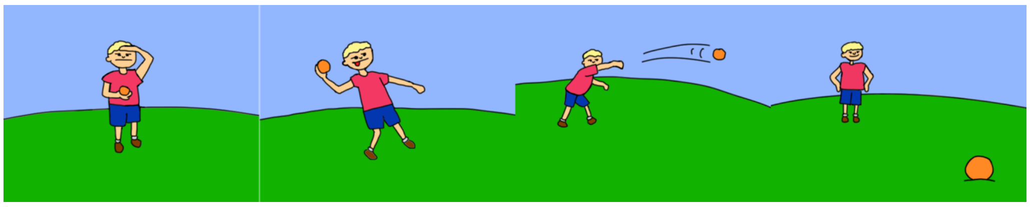


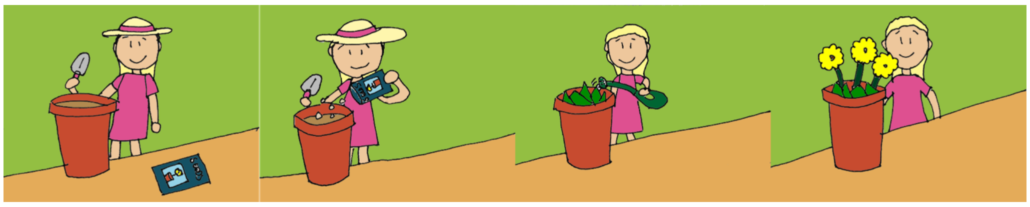
Mechanical 2 (people and objects acting causally on each other)

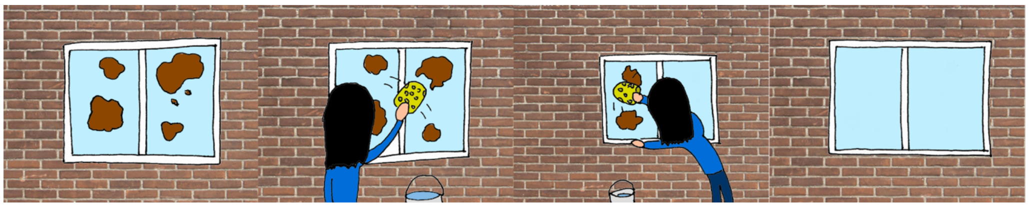




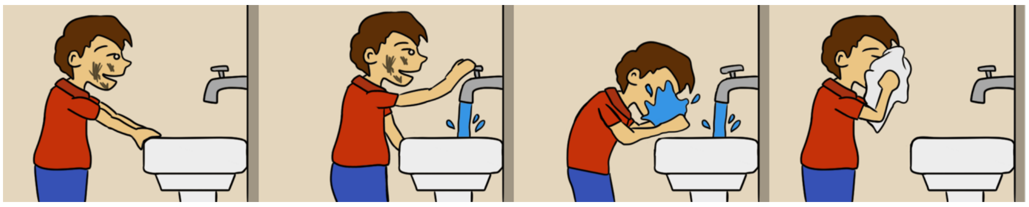


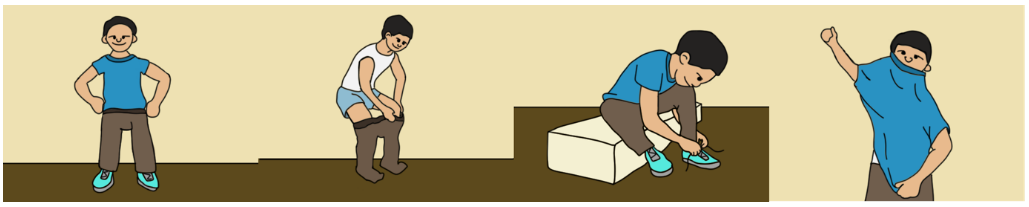


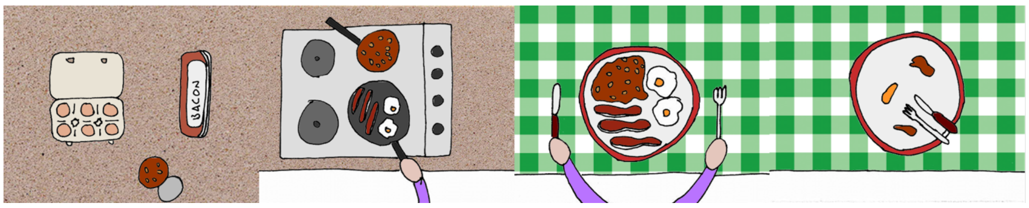


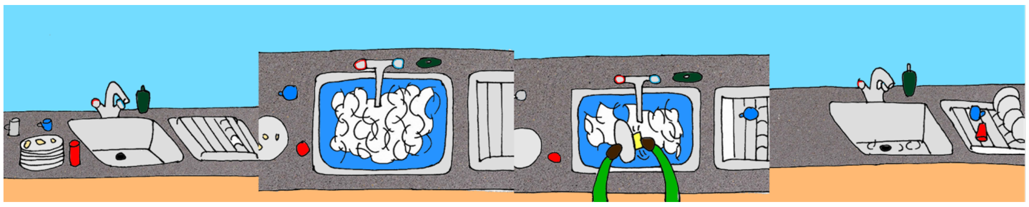


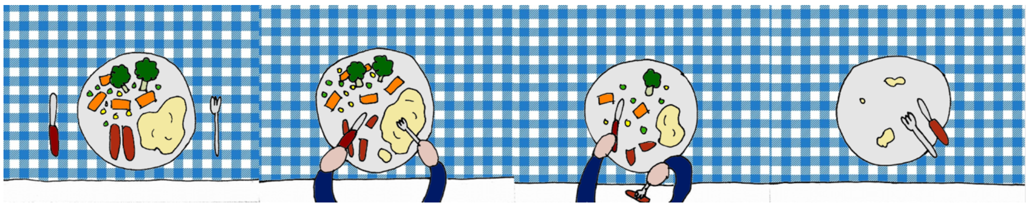
Behavioural 1 (a single person acting out everyday routines)

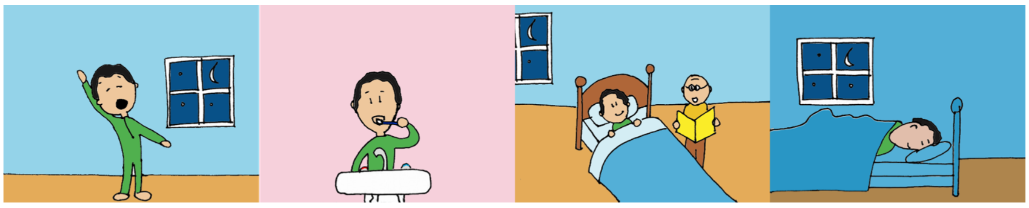




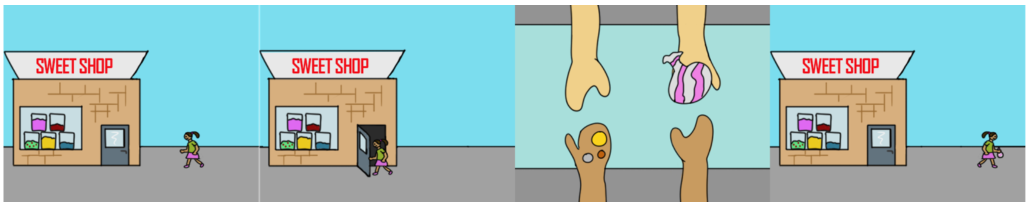


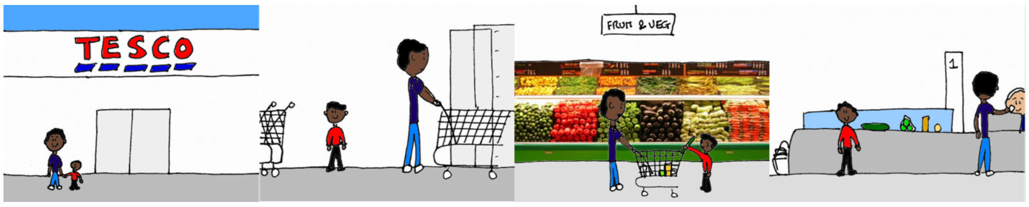


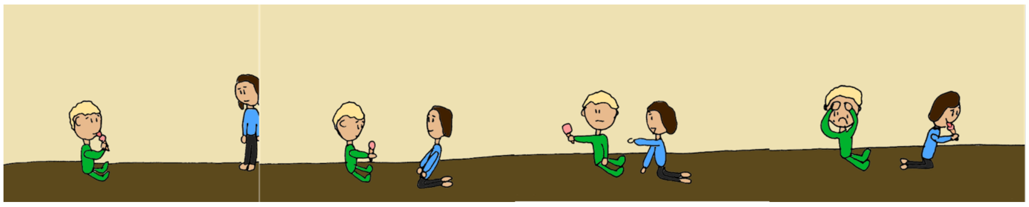


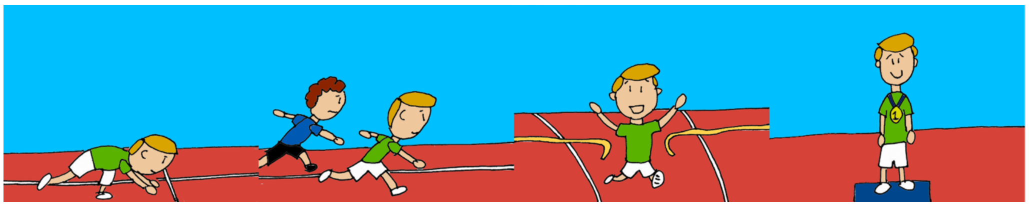


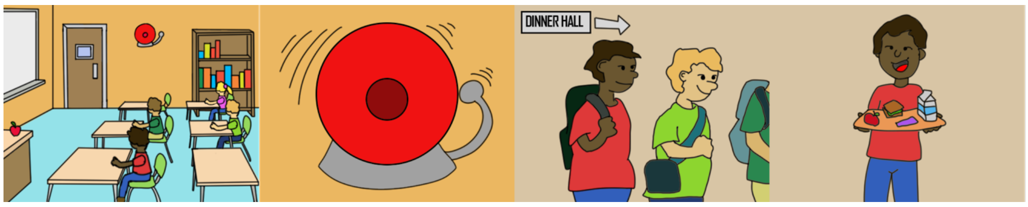
Behavioural 2 (people acting in social routines)

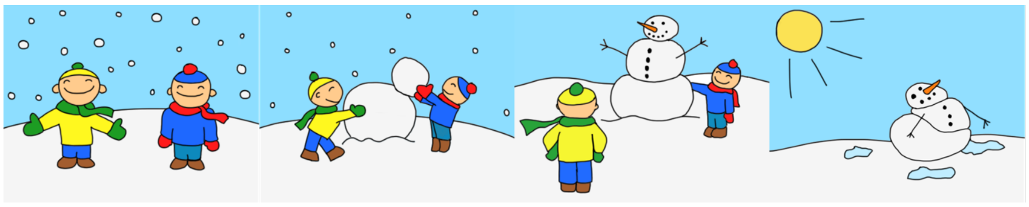




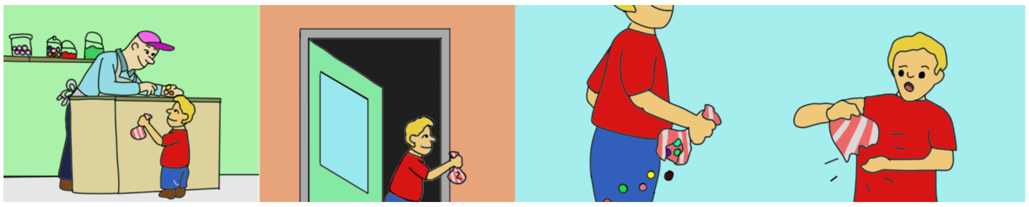


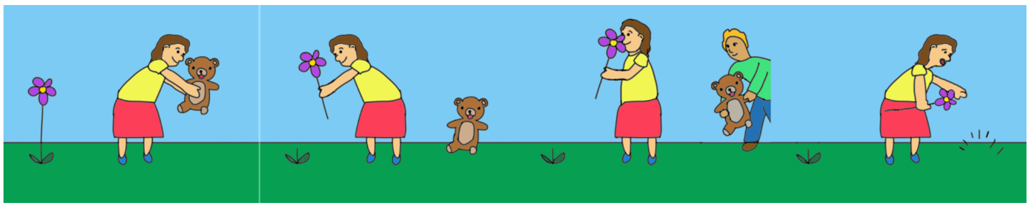


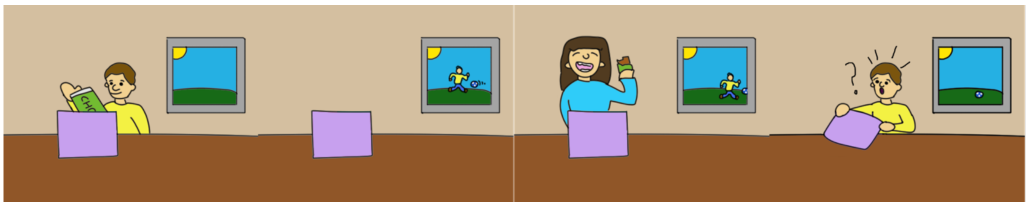


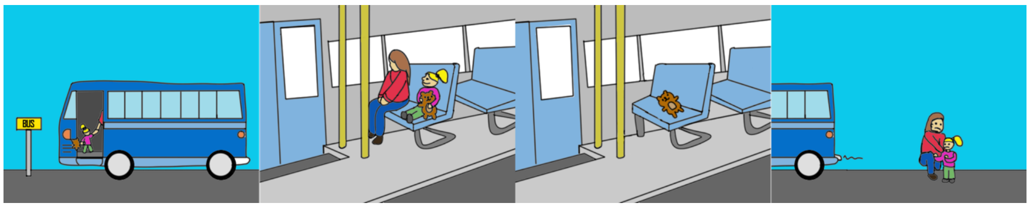


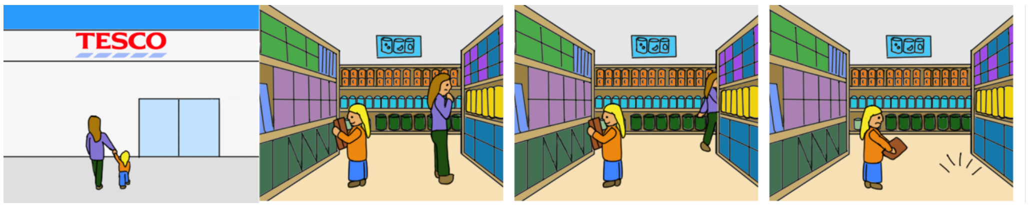
Intentional (people acting in everyday activities requiring the attribution of mental states)

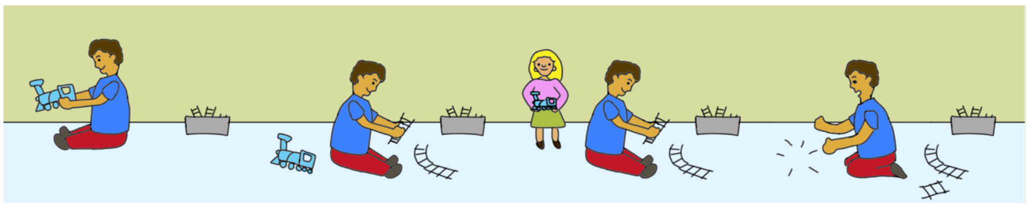












F Study 1 - Parental Consent Form

Date

Dear Parent/Carer,

In May, children in your child's class will have the opportunity to take part in a computing research study conducted by a PhD student from Sheffield Hallam University. The study will involve up to 20 minutes using a piece of software every day for a week. The software is designed to improve children's computing and problem-solving abilities in line with the national curriculum for Computing. This study is being undertaken to provide the student with research experience as part of their postgraduate programme.

To perform this research, it is necessary to make written notes on the children's experiences and test their performance using the system. This data is taken for research purposes only and is not related to the assessment of your child at their school. To preserve the anonymity of the children their real names will not be retained and will only be known temporarily by a small group of researchers involved in the project. The anonymous data may be shared with other researchers after the project is complete.

If you are happy for your children to take part in this study then there is no need to respond to this letter. However, if you DO NOT wish your child to take part in this study then please fill out the form below and return it to your child's class teacher. You or your child can also choose to withdraw from the study at any time by notifying the student researcher or your child's class teacher.

*****ONLY FILL IN THIS FORM IF YOU WISH TO OPT OUT OF THE STUDY*****

I am happy for my child **NOT** to take part in the study described above.

Name of child: ____

Class: ______________________

Signature of parent/guardian:

____

G Programming Games - Ethics Application

Designing and evaluating a programming game to improve children's abstraction skills

Proposed start date of data collection:
1/12/2017

Proposed end date of data collection:
26/09/2019

General overview of the study

Policy makers argue that children be taught how technology works, and that the 'computational thinking' (Wing, 2006) skills developed through programming are useful in a wider context (The Royal Society, 2012). This is causing an increase of computer science in primary and secondary education.

Block-based visual programming tools, like Scratch, are now being used in primary schools throughout the UK. However, Scratch users often struggle to detect and correct 'code smells' (bad programming practices) such as duplicated blocks and large scripts, which can lead to programs that are difficult to understand, debug and maintain (Hermans & Aivaloglou, 2016). These 'smells' are caused by a lack of abstraction and decomposition in programs. Scratch users can use custom blocks and clones to fix these problems, yet this functionality is not often used (Robles, Moreno-León, Aivaloglou, & Hermans, 2017).

The aim of the research is to design, develop and evaluate a programming game that teaches children to use custom blocks and clones in Scratch. The game will be tested through a series of quasi-experimental pre-test post-test studies. Participants will be 9 to 11-year-old children attending primary schools in Sheffield.

Dr Scratch (Moreno-León & Robles, 2015) analyses Scratch projects for computational thinking. It will be used to analyse participant projects created to a specification, before and after playing the programming game. Two groups will use different versions of the game; debugging and guided discovery, and an active control group will do a non-programming activity. It is expected that the participants will learn how to use custom blocks and clones in Scratch by playing the programming game. Participants will also complete a computational thinking assessment and be interviewed about their projects. This will establish whether they have learnt the underlying computational thinking skills, or have just been taught to use the blocks.

Background to the study and scientific rationale (500-750 words)

<u>Background</u>

Today's children will go on to live a life greatly influenced by computing, both in the home and at work (Barr & Stephenson, 2011). Policy makers, supported by the technology industry, are arguing that children are taught how this technology works, in order to produce 'digital citizens' for an increasingly IT-based global economy (The Royal Society, 2012). One of the main ideas behind this is that the 'computational thinking' skills developed through programming are useful in a wider context.

Computational thinking is "the conceptual foundation required to solve problems effectively and efficiently with solutions that are reusable in different contexts" (Shute, Sun, & Asbell-Clarke, 2017, p. 142). However, despite claims that computational thinking is a general skill that can be applied in other subject areas, there is little empirical support for this argument (Tedre & Denning, 2016).

Scientific rationale

Block-based visual programming tools, like Scratch, are now being used in primary schools throughout the UK. Yet, Scratch users often struggle to detect and correct 'code smells' (bad programming practices) such as duplicated blocks and large scripts, which can lead to programs that are difficult to understand, debug and maintain (Hermans & Aivaloglou, 2016). These 'smells' are caused by a lack of abstraction and decomposition in programs. Scratch users can use custom blocks and clones to fix these problems, yet this functionality is not often used (Robles, Moreno-León, Aivaloglou, & Hermans, 2017).

Dr Scratch (Moreno-León & Robles, 2015) assesses Scratch projects for CT skills. It measures abstraction and decomposition through the use of custom blocks and clones. Abstraction is the process of removing detail from a problem to generate patterns and find similarities in problems, linking closely with other similar skills; generalisation, pattern recognition and decomposition. It is a key concept in computer science and the main tenant of computational thinking (Wing, 2008). However, teaching this concept to novices is a difficult task (Armoni, 2013).

Games are also a popular method of teaching programming to novices. Programming games usually require players to navigate puzzle-based systems by programming characters using block-based or text-based instructions. These games can follow different design approaches, from open-ended exploration to linear puzzles with lots of direct guidance (Bauer, Butler, & Popović, 2017). Other games use a debugging-first approach, where the player has to debug existing broken code in order to complete levels (Lee et al., 2014).

The aim of the research is to design, develop and evaluate a programming game that teaches children to use custom blocks and clones, and therefore abstraction, in Scratch.

Main research questions

This phase of the research is focused on building and evaluating a programming game. The research questions are as follows:

- Can children be taught how to use custom blocks and clones in Scratch using a programming game?
- Does being able to use custom blocks and clones in Scratch reduce 'code smells' (bad programming habits)?
- Do improvements in using Scratch correspond to improvements in computational thinking?
- Are certain game-based learning approaches more effective than others?

Summary of methods including proposed data analyses (Include outline of techniques to be used but do not include actual protocols)

A series of formative investigations will be carried out using the programming game during an iterative design and development process. These will help to refine the level progression, the scaffolding of concept introduction, and identify any issues or bugs within the software.
The summative studies in the programme of research will follow a quasi-experimental pre-test post-test design and will be similar in terms of ethical procedure. The research design described below will then be adapted for further investigations.

The participants will be between 9 and 11 years old (year 5 and 6). They will be familiar with Scratch but will have limited knowledge of custom blocks and clones as these concepts are not taught in Sheffield's primary Computing Scheme of Work (Bush & Harris, 2014). Initial investigations have shown that children of this age can be taught to see the benefits of both custom blocks and clones in Scratch project development.

For the pre-test, participants will create an animation in Scratch from a specification. The project will involve navigating a map and 'dropping' sprites in certain locations. This task allows participants to create both simple solutions with duplicated blocks and long scripts, and be able to recognise patterns and use abstraction by using multiple sprites, custom blocks and clones. Projects will be analysed using Dr Scratch, giving each participant a score of abstraction and decomposition out of 3. Artifact-based interviews (Brennan & Resnick, 2012) will establish participants reasoning for using certain blocks. Participants will also complete a written computational thinking assessment to measure their application of computational thinking in a different context. This will give a more overall indication of computational thinking than just the completion of the design task.

Participants will then be split into three groups. The intervention group will play the programming game, one active control group will do a Scratch curriculum, and the other will do a non-programming activity in line with their educational goals. All participants will then complete similar assessments for the post-test.

The primary measure of these studies will be the pre-and post-test scores, which will be analysed to test for differences between and within groups before and after the intervention. The quantitative data will also include the computational thinking assessment scores, and interaction data recorded by the game. Qualitative data will be researcher observations, participant feedback during the intervention and the artefact-based design interviews.

The first exploratory part of this research was approved by FREC in April 2017.

Where data is collected from human participants, outline the nature of the data, details of anonymisation, storage and disposal procedures if these are required (300 - 750)

Data collected will be Scratch projects, computational thinking assessment scores, interview audio recordings and game logs. These data will be stored with a participant identification number and gender. A paper copy of names and identification numbers will be stored separately to the data, kept in the school, and disposed of after the study is complete. More detail can be found in the Data Management Plan (Appendix 4).

Describe the arrangements for recruiting, selecting/sampling and briefing potential participants. (This should clearly indicate if participants with a particular health condition or healthy volunteers are being used, the inclusion and exclusion criteria. Upload and reference copies of any advertisements for volunteers, letters to individuals/organisations inviting participation and participant information sheets. The sample sizes with power calculations if appropriate should be included)

The participants for the studies will be children aged between 9 and 11. Permission to perform each of the studies will be obtained via letters to the Headteachers of local primary schools (an example can be seen in Appendix 1).

The sample size will be minimum of two classes of approximately 30 children (60 participants in total) for each study conducted during the programme of research. They will be briefed by their teacher and again by the researchers before the study takes place (an example of the briefing given to a child by a researcher can be seen in Appendix 3).

Indicate the activities participants will be involved in. (In particular this should highlight any instances of providing biological samples, taking pharmacologically active substances or nutritional supplements, or participating in diet or exercise programmes or activities.)

Participants will be creating Scratch projects to a specification, taking a computational thinking assessment, some may be interviewed about their Scratch projects. The participants that are part of the intervention will be playing a programming game, whilst the participants in the control will be doing some other non-programming activity that fits their educational goals.

What is the potential for participants to benefit from participation in the research?

Participants may benefit from the exposure to programming tools like those used in the school curriculum. This may also positively influence their computational thinking ability and problem-solving skills, and help them work towards the computing national curriculum for Key Stage 2. The participants may also get some enjoyment from playing the programming game.

Describe any possible negative consequences of participants in the research along with the ways in which these consequences will be limited

Participants could potentially misinterpret the study as a test of themselves rather than the software, so it will be made clear at the start of the study that this is not a test of the participants and doesn't count towards their school work in any way. They will be specifically informed that it is the software we are testing and reminded that they can drop out of the study at any time.

It is expected that the studies will take place during school hours. So, there may be negative consequences in missing out on normal teaching time. To limit this, it will be ensured that the programming game and other activities are relevant to their current learning outcomes.

Describe the arrangements for obtaining participants' consent. (This should include uploads and references to the information that they will receive (participant information sheet) and participant written consent forms where appropriate. If children or vulnerable people are to be participants in the study, details of the arrangements for obtaining consent from those acting in loco parentis or as advocates should be provided.)

A parental consent form will be sent out by each school detailing each study. The form in Appendix 2 represents the starting point for this consent form, but the exact contents will be adapted per the wishes of the school. Each participant will be told (in child-friendly language) that they are taking part in a research project, it's nothing to do with their school work or assessments, and that data will be recorded but their names won't be. They will also be told that they don't have to take part and can do an alternative activity (arranged by the school), and if they have any questions they can ask at any time (see Appendix 3 for the full script).

Describe how participants will be made aware of their right to withdraw from the research. (This should also include information about participants' right to withhold information and a reasonable time span for withdrawal should be specified.)

Each child will be told in child-friendly language that they can withdraw from the study at any time. This option will be given to them when first participating in the study so they have sufficient time to decide whether they would like to participate or not.

Should a child withdraw, or parental consent is not obtained, they will either still take part in the activity, but data will not be recorded, or the school will arrange an alternative classroom activity (the wishes of the school will be followed in this regard). The game will have an option for disabling the recording of data for individual participants.

If your project requires that you work with vulnerable participants describe how you will implement safeguarding procedures during data collection

The class teacher will be present in the room throughout the study. The school safeguarding procedures will be followed and any problems will be deferred to the teacher (such as behavioural issues).

If Disclosure and Barring Service (DBS) checks are required, please supply details

The principle investigator has a valid DBS certificate carried out by Southey Green Community Primary School in October 2015. Certificate number: 001505136041

Describe the arrangements for debriefing the participants

Each child will be thanked for their contribution and asked if they have any feedback to give. It will be reaffirmed that the study has nothing to do with their school work or assessments and all their answers are anonymous.

Describe the arrangements for ensuring participant confidentiality.

* how data will be stored to ensure compliance with data protection legislation

* how results will be presented

* exceptional circumstances where confidentiality may not be preserved

* how and when confidential data will be disposed of

* reference to the data management plan in relation to archiving etc

Data will be stored with an assigned identification number. A paper copy of names and identification numbers will be stored separately to the data. This will be kept on the school site at all times and disposed of after data collection is complete. Interview recordings will be transcribed with altered names if necessary (full detail of this can be seen in the data management plan in Appendix 4). The results will be presented in the form of a thesis and research papers and will be anonymous. There are no exceptional circumstances in which confidentiality will not be preserved.

Are there any conflicts of interest in you undertaking this research?

None.

What are the expected outcomes, impacts and benefits of the research?

The expected outcome of the research is that the programming game will improve participants' knowledge of Scratch, specifically their ability to use custom blocks and clones. It may also improve their wider computational thinking ability.

Overall, it is hoped that the research will make a valuable contribution to the fields of game-based learning and computational thinking.

Please give details of any plans for dissemination of the results of the research. (This includes your plans for preserving and sharing your data. You may refer to your attached Data Management Plan.)

It is hoped the research will be published as part of a PhD thesis and as research papers. Data will be stored in the SHURDA, and made available to other institutions upon request (see Appendix 4 for more detail).

H Programming Games - Data Management Plan

Project Name Designing and evaluating a programming game to improve children's abstraction skills
Principal Investigator / Researcher Simon Rose
Institution Sheffield Hallam University

Data collection
What data will be produced?
Quantitative data in the form of scores from a Scratch project analysis tool, scores from a computational thinking assessment, and interaction logs from a game. The scores from the tool and assessment will be stored in an Excel spreadsheet. The interaction logs stored digitally in a MongoDB database in BSON format. MongoDB works well with the language the game will be written in (JavaScript) and is fast and easy to maintain. BSON is a lightweight file format (based on JSON) that does not take up much storage space (MB's). The raw data can be exported as JSON or CSV, or to SPSS for storage on SHURDA or the university research drive. MongoDB also handles file versioning. The data will be analysed using Node.js to extract participant data, which will be stored in an SPSS file.

Qualitative data in the form of written observations, participant feedback and interview recordings. This will be collected as sound files and notes that are written up in rich text format (rtf).

Data documentation
How will your data be documented and described?
The quantitative data will be documented using metadata. It will be stored in a database that will have suitable column names that should be readable to someone who is not associated with the project. The SPSS file that the data will be analysed using will be documented in a readme text file, and have understandable column names. The observation notes will be written up and accompanied by a readme to describe ambiguous comments and structure.

Ethical and copyright issues
How will you deal with any ethical and copyright issues?
As the participants are children, consent will be obtained from their parents or carers. This consent form will clarify that parents are giving consent for the researchers to collect data and that this data may be shared with other institutions.

All data will be anonymised and assigned an ID number that is user dependent. Names recorded in the observation notes will be changed before they are stored. Audio recordings will be deleted once transcribed to avoid the possibility of participant identification.

The copyright of the data will lie with the University, and it will be stored in SHURDA once collected.

Data storage

How will your data be structured, stored and backed up?

Data will be recorded in an Excel spreadsheet, or for the game, logged to a MongoDB database from a JavaScript web application. It will be sent from the application to a NoSQL database on a cloud platform (such as DigitalOcean). Backups of the data will be placed on the University research drive. Study observations and audio recordings will be written up as soon as possible and stored on the research drive until the study is complete. These observations will follow a filename format that includes the date and session in which they were recorded.

Data preservation

What are the plans for the long-term preservation of data supporting your research?

The quantitative data will be stored long-term, as it is anonymised and will not need to be deleted on any contractual grounds. It is hoped that the data will be used in a research publication, and can be used to facilitate further analysis.

The data will be stored in the SHURDA upon completion of the study.

Data sharing

What are your plans for data sharing after submission of your thesis?

It is hoped that the data will be useful to other researchers and can be further analysed. The data will be stored in the SHURDA, and be given to other institutions upon request.

I Study 3 - Pirate Plunder Debugging-First Instructions

Table 1: Pirate Plunder challenge progression including debugging-first blocks

Required Tutorial	Challenge Number	Coins	Maximum Block Count	Debugging Blocks				
				Incorrect Input	Locked Input	Assistance	Incorrect Block	Total
Go to position	1	0	3	1	0	1	0	2
	2	1	4	1	0	0	0	1
Move	3	1	4	1	0	1	2	4
	4	3	6	2	1	0	0	3
	5	6	9	2	2	0	0	4
Turn	6	1	5	2	1	1	0	4
	7	1	6	2	1	0	0	3
	8	7	11	1	2	0	1	4
	9	7	13	0	3	1	0	4
	10	13	18	4	0	0	0	4
Repeat	11	2	4	2	0	0	0	2
	12	3	5	2	1	0	0	3
	13	8	7	3	0	0	0	3
	14	11	10	2	2	0	1	5

Continued on next page

Continued from previous page

Required Tutorial	Challenge Number	Coins	Maximum Block Count	Debugging Blocks				
				Incorrect Input	Locked Input	Assistance	Incorrect Block	Total
	15	18	17	3	3	0	1	7
Show/hide	16	1	7	0	0	4	1	5
	17	8	9	2	0	1	0	3
	18	7	11	2	2	0	0	4
Custom blocks	19	11	9	1	1	1	1	4
	20	14	10	1	1	1	0	3
	21	14	11	2	0	2	0	4
	22	14	14	1	2	2	0	5
	23	15	16	0	0	2	4	6
	24	20	17	0	0	1	4	5
Inputs	25	14	10	0	0	3	0	3
	26	13	10	2	0	1	1	4
	27	8	11	1	1	0	1	3
	28	15	11	2	0	1	0	3
	29	8	10	1	1	2	0	4
	30	15	10	0	0	0	1	1
Cloning	31	5	16	6	2	4	0	12

Continued on next page

Continued from previous page

Required Tutorial	Challenge Number	Coins	Maximum Block Count	Debugging Blocks				
				Incorrect Input	Locked Input	Assistance	Incorrect Block	Total
	32	4	18	5	0	4	0	9
	33	9	19	5	0	7	0	12
	34	9	20	3	0	2	1	6
	35	9	18	2	0	2	0	4
	36	15	18	1	0	4	0	5
	37	13	20	0	0	2	1	3
	38	7	20	0	0	2	1	3
	39	11	20	1	0	0	1	2
	40	19	26	0	0	0	0	0

J Scratch Task - Planting Trees

Scratch Task – Planting Trees

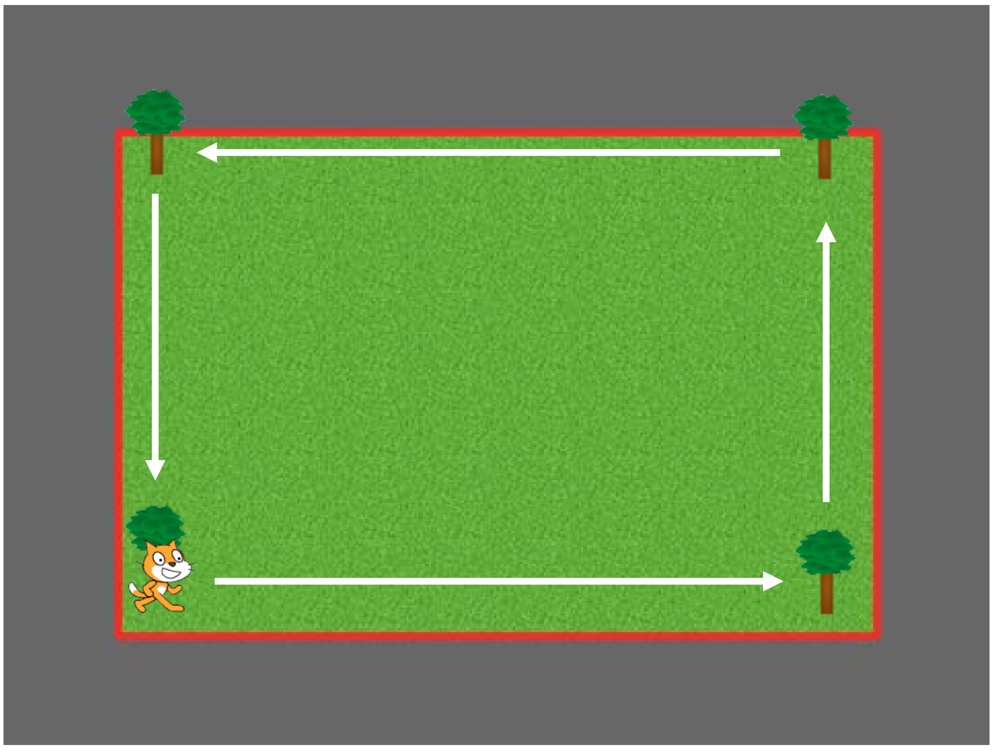
This park needs some trees! Can you create a Scratch project that does the following?

When the green flag is pressed, animate Scratch cat's movement around the edge of the park, planting a tree in each corner.

There are lots of ways of doing this, try to come up with the solution that you think is best. There are some blocks already in the project that are there to help you out.

When you've finished, 'save as' your attempt as a version and see if you can improve your solution, either by using different blocks or by adding some extra functionality.

You have this session to come up with your best solution. Good luck!



K Scratch Task - Setting Up Speakers

Scratch Task – Setting Up Speakers

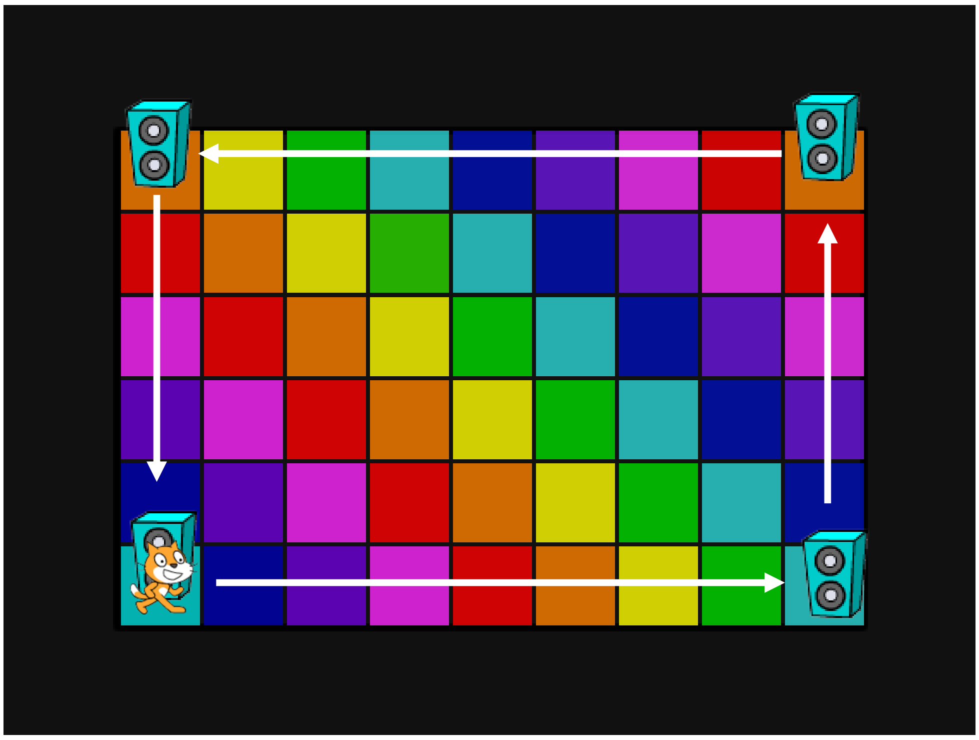
This room needs speakers for a party! Can you create a Scratch project that does the following?

When the green flag is pressed, animate Scratch cat's movement around the edge of the room, putting a speaker in each corner.

There are lots of ways of doing this, try to come up with the solution that you think is best. There are some blocks already in the project that are there to help you out.

When you've finished your attempt, 'save as' it as a version and see if you can improve your solution, either by using different blocks or adding some extra functionality.

You have this session to come up with your best solution. Good luck!



L Artifact-Based Interview Script

Interview Script

I'd like to ask you a few questions about the Scratch project that you made yesterday. It won't take very long. Do you mind if I record audio of the interview so that I can make notes about it later?

Firstly, can you explain in detail how your project works?

What do each of the blocks do?

Where did you learn to do that?

What specifically?

Are there any advantages to doing it this way?

Are there any disadvantages to doing it this way?

Could you have done it another way?

Could you have used a custom block?

Why would you use a custom block?

Can you give another example of how you'd use a custom block?

Could you have used cloning on the tree/speaker?

Why would you have used cloning?

Can you give another example of how you'd use a clone?

Are there any similarities between this task and Pirate Plunder?

What are they?

Could you have solved the problem in the same way that you completed levels in Pirate Plunder?

Can you elaborate?

M Study 3 - Pirate Plunder Questionnaire

Name: Class:

Please answer the questions honestly.

How confident do you feel using Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using Scratch changed after playing Pirate Plunder?

🙂	😕	☹
Improved	Same as before	Declined

How confident do you feel using **custom blocks** in Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using **custom blocks** changed after playing Pirate Plunder?

🙂	😕	☹
Improved	Same as before	Declined

How confident do you feel using **clones** in Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using **clones** changed after playing Pirate Plunder?

🙂	😕	☹
Improved	Same as before	Declined

Please turn over

What did you like about Pirate Plunder?

Is there anything that you would change about Pirate Plunder? (problems, difficulties, extra features)

Do you have any other comments about Pirate Plunder or about the IT sessions in general?

N Study 3 - Scratch Questionnaire

Name:	Class:

Please answer the questions honestly.

How confident do you feel using Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using Scratch changed after the Scratch lessons?

🙂	😕	☹
Improved	Same as before	Declined

How confident do you feel using **custom blocks** in Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using **custom blocks** changed after the Scratch lessons?

🙂	😕	☹
Improved	Same as before	Declined

How confident do you feel using **clones** in Scratch?

😀	🙂	😕	☹
Very confident	Confident	Slightly confident	Not confident

How has your confidence using **clones** changed after the Scratch lessons?

🙂	😕	☹
Improved	Same as before	Declined

Please turn over

What did you like about the Scratch lessons?

Is there anything that you would change about the Scratch lessons?

Do you have any other comments?

O Study 3 - Parental Consent Form

Date

Dear Parent/Carer,

Post SATs, children in your child's class will have the opportunity to take part in a computing research study led by Simon Rose, a PhD student from Sheffield Hallam University. The study will involve playing a programming game that Simon has created for his research, which is designed to improve children's computing and problem-solving abilities in line with the national curriculum for Computing. You can contact Simon if you have any concerns at simon.p.rose@student.shu.ac.uk.

To perform this research, it is necessary to assess their programming and computational thinking abilities, store anonymous analytics from the game, make written notes on the children's experiences and conduct audio recorded interviews (which will be transcribed and anonymised). This data is taken for research purposes only and is not related to the assessment of your child at their school. To preserve the anonymity of the children their real names will not be retained and will only be known temporarily by a small group of researchers involved in the project. The anonymous data may be shared with other researchers after the project is complete.

If you are happy for your child to take part in this study, then **there is no need to respond to this letter**. However, if you **DO NOT** wish your child to take part in this study then please fill out the form below and return it to your child's class teacher. You or your child can also choose to withdraw from the study at any time by notifying the student researcher or your child's class teacher.

*****ONLY FILL IN THIS FORM IF YOU WISH TO OPT OUT OF THE STUDY*****

I would prefer my child **NOT** to take part in the study described above.

Name of child: ____

Class: ______________________

Signature of parent/guardian:

____

P Study 4 - Pirate Plunder Revised Debugging-First Instructions

Table 2: Pirate Plunder Study 4 level progression

Required Tutorial	Challenge Number	Coins	Maximum Block count	Old Debugging Blocks	New Debugging Blocks
Go to position	1	0	3	2	0
	2	1	4	1	0
Move	3	1	4	4	0
	4	3	6	3	3
	5	6	9	4	2
Turn	6	1	5	4	1
	7	1	6	3	1
	8	7	11	4	3
	9	7	13	4	3
	10	13	18	4	4
Repeat	11	2	4	2	1
	12	3	5	3	0
	13	8	7	3	0
	14	11	10	5	0
	15	18	17	7	0
Show/hide	16	1	7	5	2
	17	8	9	3	0

Continued on next page

Continued from previous page

Required Tutorial	Challenge Number	Coins	Maximum Block Count	Old Debugging Blocks	New Debugging Blocks
	18	7	11	4	0
Custom blocks	19	11	9	4	5
	20	14	10	3	0
	21	14	12	4	1
	22	15	14	5	3
	23	14	14	6	5
	24	20	17	5	5
Inputs	25	14	10	3	3
	26	13	10	4	0
	27	8	11	3	1
	28	15	11	3	0
	29	8	10	4	4
	30	15	10	1	0
Cloning	31	5	16	12	13
	32	4	18	9	10
	33	9	19	12	7
	34	9	20	6	2
	35	9	18	4	3
	36	15	18	5	5
	37	13	20	3	0

Continued on next page

Continued from previous page

Required Tutorial	Challenge Number	Coins	Maximum Block Count	Old Debugging Blocks	New Debugging Blocks
	38	7	20	3	0
	39	11	20	2	0
	40	19	26	0	0

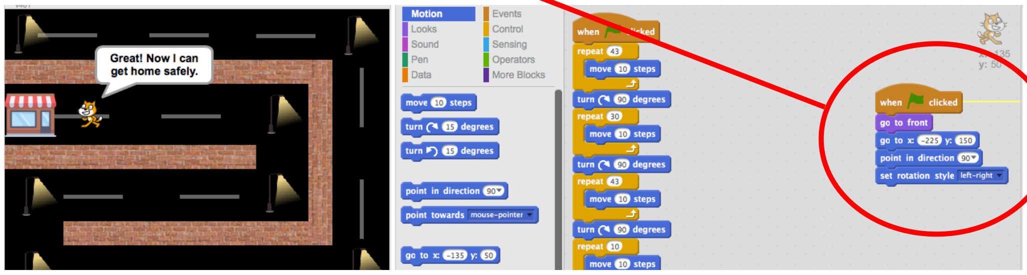
Q Scratch Challenge

Scratch Challenge

In this project, Scratch cat goes around a maze of dark streets, so that he can get to the shop to buy some food. He is leaving lampposts along the way (at the start and on each corner) so that he doesn't have to go back in the dark!

Your aim is to produce the same result with fewer blocks and fewer sprites. The lower the number of blocks the higher your score. But make sure you follow the rules:

- **You must not use** `glide ( ) secs to x: ( ) y: ( )` or `go to x: ( ) y: ( )` **in the Cat** (but you can use them in the lampposts)
- **Leave in** the "Great! Now I can get home safely." speech block.
- **Do not delete** these blocks



```
move (10) steps
turn ↻ (15) degrees
turn ↺ (15) degrees

point in direction (90▾)
point towards [mouse-pointer ▾]

go to x: (-135) y: (50)
```

```
when ⚑ clicked
repeat (43)
    move (10) steps
turn ↻ (90) degrees
repeat (30)
    move (10) steps
turn ↻ (90) degrees
repeat (43)
    move (10) steps
turn ↻ (90) degrees
repeat (10)
    move (10) steps
```

```
when ⚑ clicked
go to front
go to x: (-225) y: (150)
point in direction (90▾)
set rotation style [left-right ▾]
```

(They reset Scratch cat's position and direction before each run and **will not count towards your block count**.)

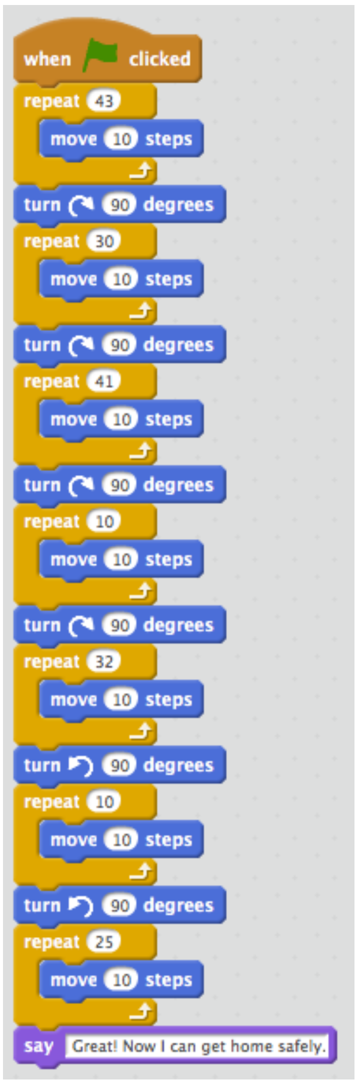
The starter project has 50 blocks in total, **which you can see on the back of this sheet.**

When you've finished your attempt, 'save as' it as a version and see if you can improve your solution.

You have the rest of this session to do this, good luck!

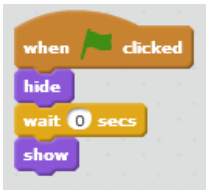
Starting blocks

Cat:



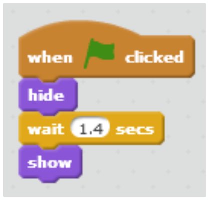
```
when 🏁 clicked
repeat 43
    move 10 steps
turn ↻ 90 degrees
repeat 30
    move 10 steps
turn ↻ 90 degrees
repeat 41
    move 10 steps
turn ↻ 90 degrees
repeat 10
    move 10 steps
turn ↻ 90 degrees
repeat 32
    move 10 steps
turn ↺ 90 degrees
repeat 10
    move 10 steps
turn ↺ 90 degrees
repeat 25
    move 10 steps
say Great! Now I can get home safely.
```

Lamppost1:



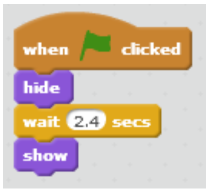
```
when 🏁 clicked
hide
wait 0 secs
show
```

Lamppost2:



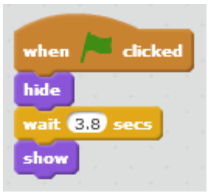
```
when 🏁 clicked
hide
wait 1.4 secs
show
```

Lamppost3:



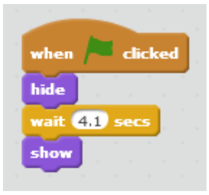
```
when 🏁 clicked
hide
wait 2.4 secs
show
```

Lamppost4:



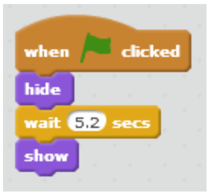
```
when 🏁 clicked
hide
wait 3.8 secs
show
```

Lamppost5:



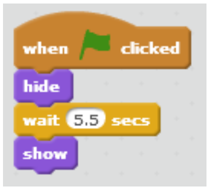
```
when 🏁 clicked
hide
wait 4.1 secs
show
```

Lamppost6:



```
when 🏁 clicked
hide
wait 5.2 secs
show
```

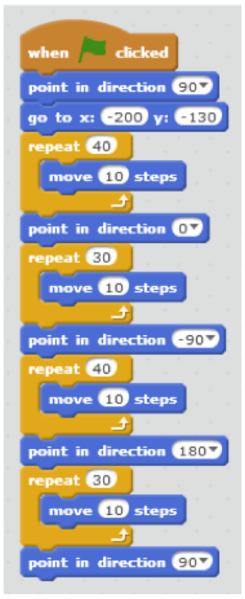
Lamppost7:



```
when 🏁 clicked
hide
wait 5.5 secs
show
```

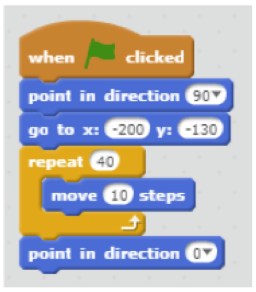
Which **ONE** of these scripts could we use a custom block to shorten?

A



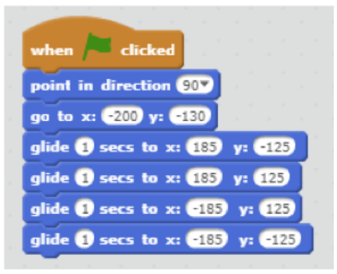
```
when ⚑ clicked
point in direction 90
go to x: -200 y: -130
repeat 40
    move 10 steps
point in direction 0
repeat 30
    move 10 steps
point in direction -90
repeat 40
    move 10 steps
point in direction 180
repeat 30
    move 10 steps
point in direction 90
```

B



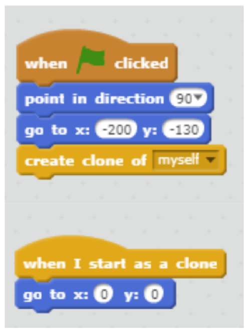
```
when ⚑ clicked
point in direction 90
go to x: -200 y: -130
repeat 40
    move 10 steps
point in direction 0
```

C



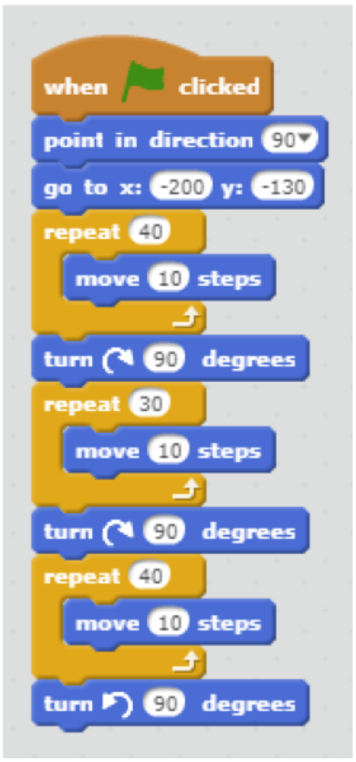
```
when ⚑ clicked
point in direction 90
go to x: -200 y: -130
glide 1 secs to x: 185 y: -125
glide 1 secs to x: 185 y: 125
glide 1 secs to x: -185 y: 125
glide 1 secs to x: -185 y: -125
```

D



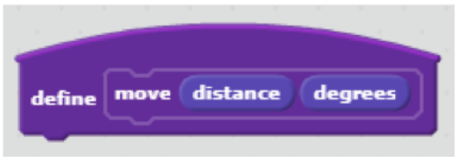
```
when ⚑ clicked
point in direction 90
go to x: -200 y: -130
create clone of myself

when I start as a clone
go to x: 0 y: 0
```

Which **ONE** of these custom block definitions would be best for this script?



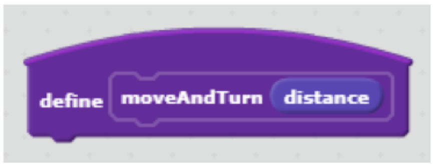
```
when ⚑ clicked
point in direction 90
go to x: -200 y: -130
repeat 40
    move 10 steps
turn ↻ 90 degrees
repeat 30
    move 10 steps
turn ↻ 90 degrees
repeat 40
    move 10 steps
turn ↺ 90 degrees
```

A



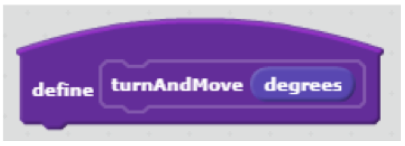
```
define move distance degrees
```

B



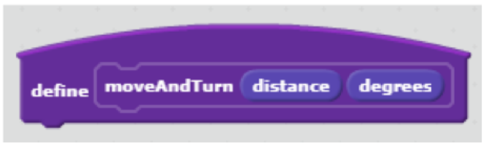
```
define moveAndTurn distance
```

C



```
define turnAndMove degrees
```

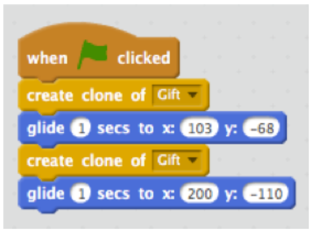
D



```
define moveAndTurn distance degrees
```

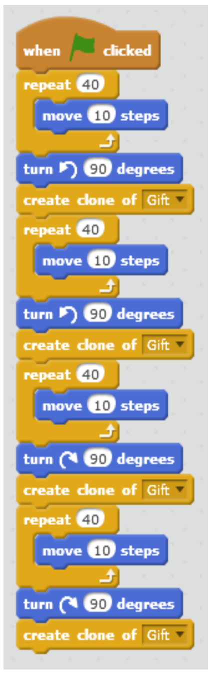
Which **ONE** of these scripts could we use a custom block to shorten?

A



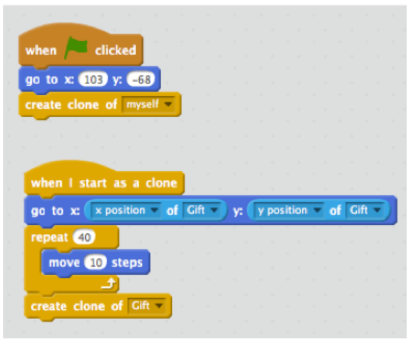
```
when ⚑ clicked
create clone of Gift ▾
glide 1 secs to x: 103 y: -68
create clone of Gift ▾
glide 1 secs to x: 200 y: -110
```

B



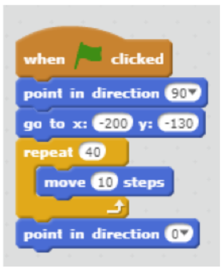
```
when ⚑ clicked
repeat 40
    move 10 steps
turn ↺ 90 degrees
create clone of Gift ▾
repeat 40
    move 10 steps
turn ↺ 90 degrees
create clone of Gift ▾
repeat 40
    move 10 steps
turn ↻ 90 degrees
create clone of Gift ▾
repeat 40
    move 10 steps
turn ↻ 90 degrees
create clone of Gift ▾
```

C



```
when ⚑ clicked
go to x: 103 y: -68
create clone of myself ▾

when I start as a clone
go to x: (x position ▾ of Gift ▾) y: (y position ▾ of Gift ▾)
repeat 40
    move 10 steps
create clone of Gift ▾
```

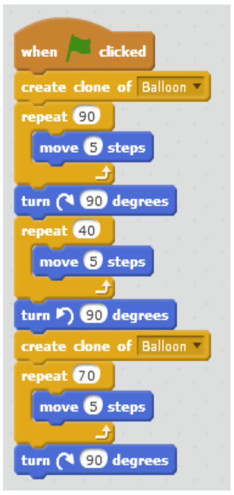
D



```
when ⚑ clicked
point in direction 90▾
go to x: -200 y: -130
repeat 40
    move 10 steps
point in direction 0▾
```

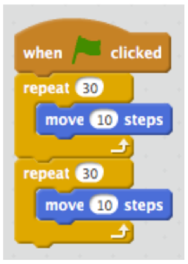
Which **ONE** of these scripts would a custom block **NOT** shorten?

A



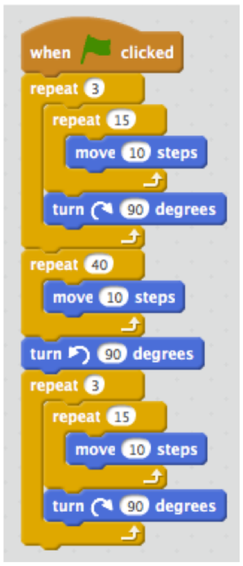
```
when ⚑ clicked
create clone of Balloon ▾
repeat 90
    move 5 steps
turn ↻ 90 degrees
repeat 40
    move 5 steps
turn ↺ 90 degrees
create clone of Balloon ▾
repeat 70
    move 5 steps
turn ↻ 90 degrees
```

B



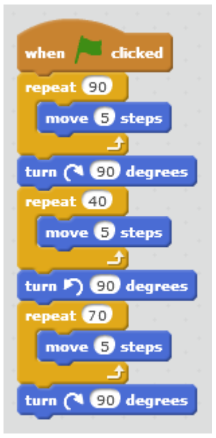
```
when ⚑ clicked
repeat 30
    move 10 steps
repeat 30
    move 10 steps
```

C



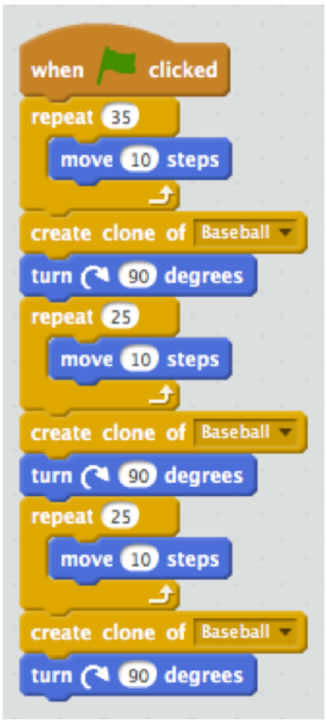
```
when ⚑ clicked
repeat 3
    repeat 15
        move 10 steps
    turn ↻ 90 degrees
repeat 40
    move 10 steps
turn ↺ 90 degrees
repeat 3
    repeat 15
        move 10 steps
    turn ↻ 90 degrees
```

D



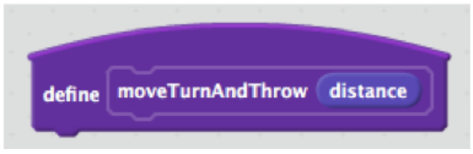
```
when ⚑ clicked
repeat 90
    move 5 steps
turn ↻ 90 degrees
repeat 40
    move 5 steps
turn ↺ 90 degrees
repeat 70
    move 5 steps
turn ↻ 90 degrees
```

Which **ONE** of these custom block definitions would be best for this script?



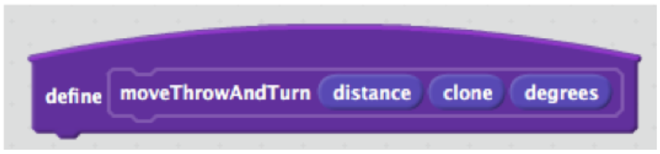
```
when green flag clicked
repeat 35
    move 10 steps
create clone of Baseball
turn ↻ 90 degrees
repeat 25
    move 10 steps
create clone of Baseball
turn ↻ 90 degrees
repeat 25
    move 10 steps
create clone of Baseball
turn ↻ 90 degrees
```

A



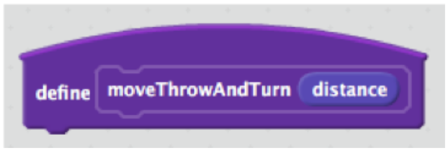
```
define moveTurnAndThrow (distance)
```

B



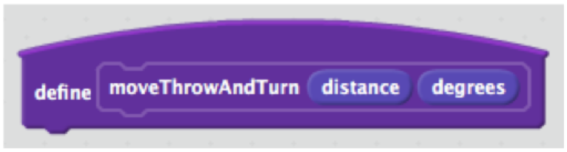
```
define moveThrowAndTurn (distance) (clone) (degrees)
```

C



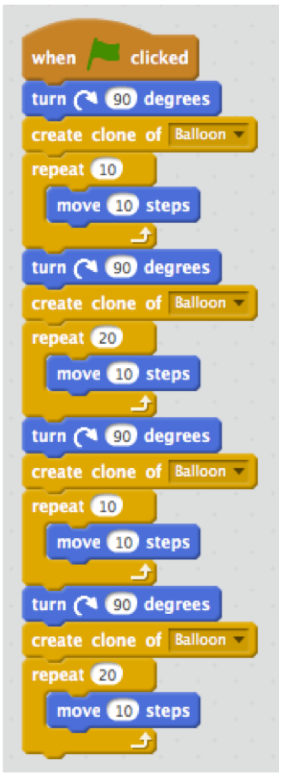
```
define moveThrowAndTurn (distance)
```

D



```
define moveThrowAndTurn (distance) (degrees)
```

How many inputs will the best custom block for this script need?

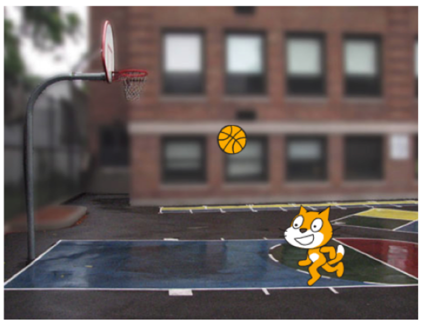


```
when [flag] clicked
turn ↻ (90) degrees
create clone of [Balloon ▼]
repeat (10)
  move (10) steps
end
turn ↻ (90) degrees
create clone of [Balloon ▼]
repeat (20)
  move (10) steps
end
turn ↻ (90) degrees
create clone of [Balloon ▼]
repeat (10)
  move (10) steps
end
turn ↻ (90) degrees
create clone of [Balloon ▼]
repeat (20)
  move (10) steps
end
```

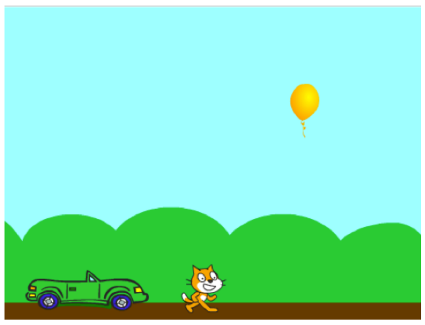
A	0
B	1
C	2
D	3

In which **ONE** of these scenes would it be worthwhile to clone one of the sprites?

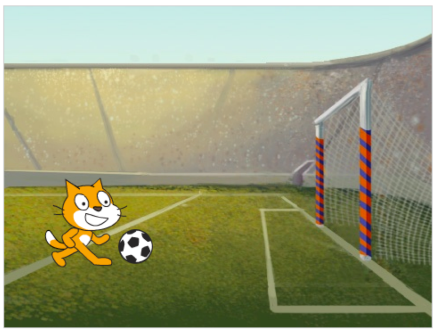
A



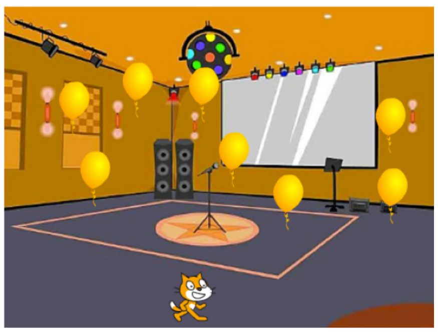
B



C

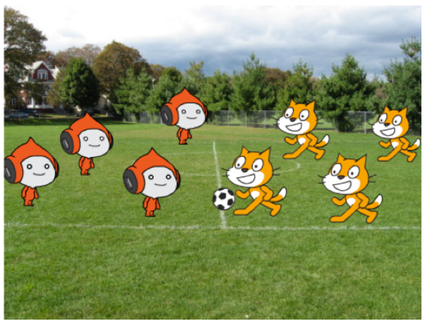


D

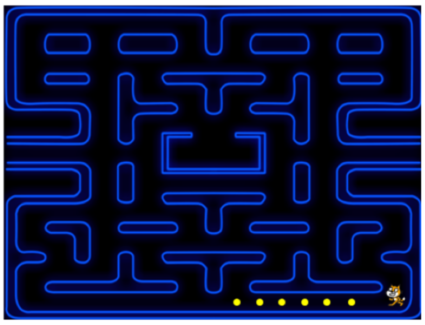


In which **ONE** of these scenes would cloning **NOT** be worthwhile?

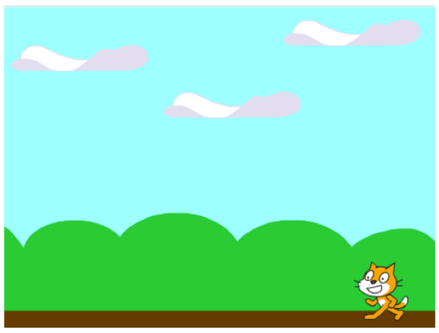
A



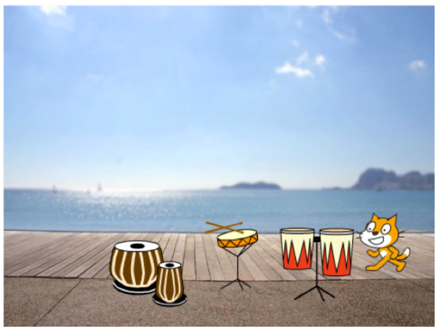
B



C

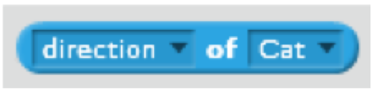


D

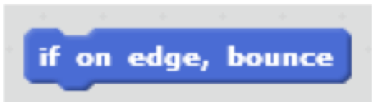


Which **ONE** of these blocks can be altered to get the x position of a sprite?

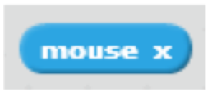
A



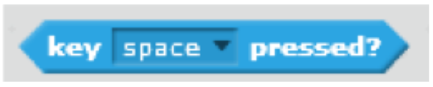
B



C

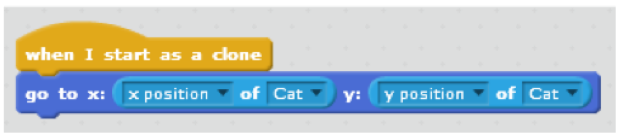


D

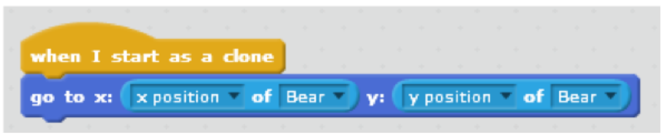


Which **ONE** of these scripts will create the clone at same x and y position as the 'Cat' sprite?

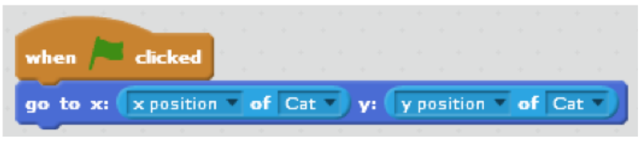
A



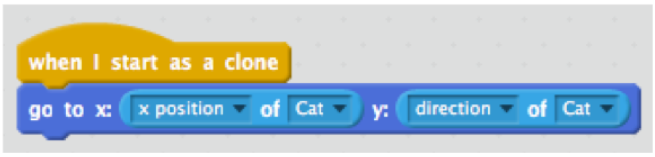
B



C



D



S Multiple-Choice Scratch Abstraction Test - Answers and Rationale

Table 4: Multiple-choice Scratch abstraction test question breakdown

	Question	Correct Answer	Rationale
1	Which **ONE** of these scripts could we use a custom block to shorten?	A	Identifying duplicated code where the block count can be reduced using a custom block.
2	Which **ONE** of these custom block definitions would be best for this script?	D	Identifying the correct block name and inputs for a set of duplicated blocks.
3	Which **ONE** of these scripts could we use a custom block to shorten?	B	Identifying duplicated code containing cloning where the block count can be reduced using a custom block.
4	Which **ONE** of these scripts would a custom block **NOT** shorten?	B	Recognising that using a custom block for blocks only duplicated twice will not reduce the project's block count.
5	Which **ONE** of these custom block definitions would be best for this script?	C	Identifying the correct block name and inputs for a set of duplicated blocks that uses cloning.
6	How many inputs will the best custom block for this script need?	B	Identifying the correct number of inputs for a set of duplicated blocks.
7	In which **ONE** of these scenes would it be worthwhile to clone one of the sprites?	D	Recognising the scene where there are multiple sprites performing the same action.
8	In which **ONE** of these scenes would it cloning **NOT** be worthwhile?	D	Recognising the scene where there are single sprites performing different actions.
9	Which **ONE** of these blocks can be altered to get the x position of a sprite?	A	Identifying the correct 'get property of' block for getting the x position of a sprite.
10	Which **ONE** of these scripts will create the clone at same x and y position as the 'Cat' sprite?	A	Identifying the correct blocks for cloning at the position of another sprite.

T Study 4 - Parental Consent Forms

Date

Dear Parent/Guardian,

This term, children in your child's class will have the opportunity to take part in a computing research study led by Simon Rose, a PhD student from Sheffield Hallam University. The study will involve playing a programming game (Pirate Plunder) that Simon has created for his research, which is designed to improve children's computing and problem-solving abilities in line with the national curriculum for Computing.

Due to changes in school policy, we now require your consent for your child to take part in this study. An information sheet has also been provided. The study will begin on Monday 8th October, so **YOUR CHILD WILL NOT BE ABLE TO TAKE PART IF YOU RESPOND AFTER THIS DATE**.

If you are happy for your child to take part in this study, please **FILL OUT AND RETURN THE CONSENT FORM BELOW**. You or your child can choose to withdraw from the study at any time by notifying the student researcher or your child's class teacher.

I am happy for my child to take part in the study described above.

Name of child: ____

Class: ___________________________

Signature of parent/guardian:

____

pirate plunder

U Study 4 - Information Sheet

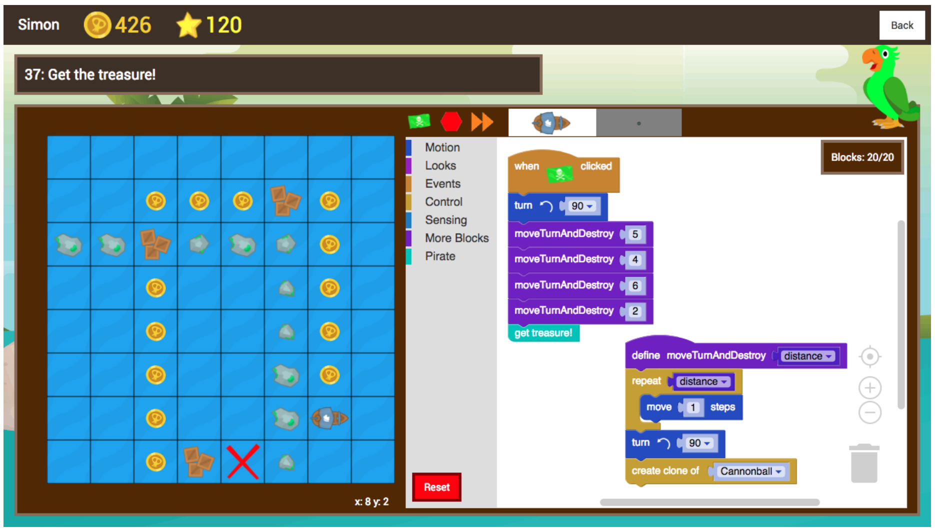
ICT Research Study - Information Sheet

Research project title:
Designing and evaluating a programming game to improve children's abstraction skills

Principal investigator:
Simon Rose
PhD student, Sheffield Hallam University
simon.p.rose@student.shu.ac.uk

About the project
Children are now being taught programming from Year 1, as outlined by the English national curriculum for computing. However, whilst the programming tools and teaching techniques used are a good introduction to computer science, they can cause bad programming practices. These can then manifest themselves when children move to text-based languages such as Python in secondary school.

One of the most widely-used programming tools is Scratch, in which users can create stories, animations and games using block-based programming. Pirate Plunder is a programming puzzle game that aims to introduce abstraction techniques in Scratch, which allow children to identify and correct bad programming practices. This is done using custom blocks (procedures) and clones (instances of sprites).



```
when clicked
turn 90
moveTurnAndDestroy 5
moveTurnAndDestroy 4
moveTurnAndDestroy 6
moveTurnAndDestroy 2
get treasure!

define moveTurnAndDestroy distance
repeat distance
    move 1 steps
turn 90
create clone of Cannonball
```

An example level from Pirate Plunder that shows what each child should be able to do by the conclusion of the study.

Sheffield Hallam University | Cultural, Communication and Computing Research Institute

What is involved in the study?

The study uses a crossover design, which means that the children will be doing two different 6-hour interventions during the course of the study. This will be:

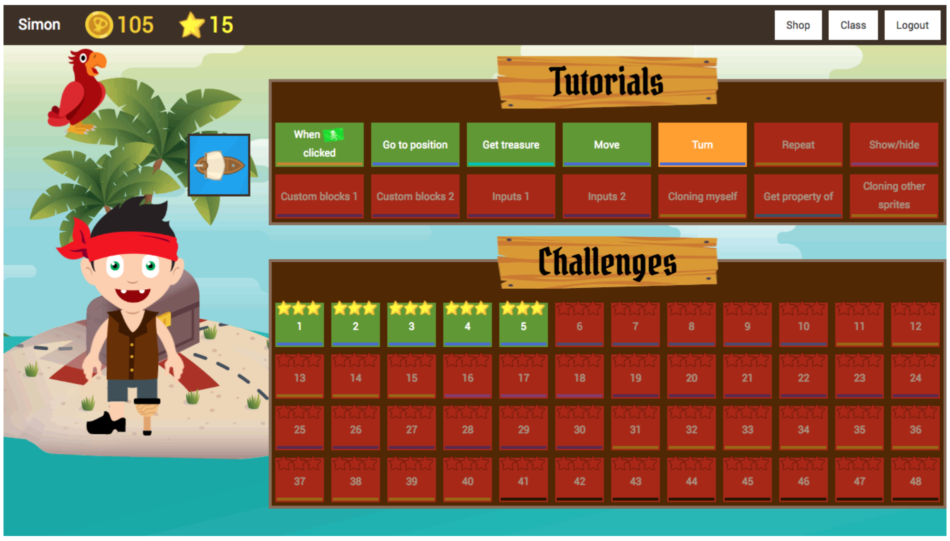
1. Pirate Plunder
2. Spreadsheets or Scratch curriculum

The children will also be assessed for their ability in Scratch, use of abstraction techniques in Scratch and their computational thinking skills before and after playing Pirate Plunder. This will be done using the following assessments:

- Scratch task (completing a Scratch project to a specification)
- Scratch multiple choice test (multiple choice on using taught concepts)
- Computational Thinking test (multiple choice)

Questionnaires and interviews will also be used as part of the study. The questionnaires are designed to gauge child confidence in Scratch and the concepts that have learnt playing Pirate Plunder. The interviews are designed to see if children have understood what they have learnt.

The overall aim of the study is to see whether the game is effective in comparison to a standard ICT curriculum.



The Pirate Plunder level select screen – player avatars can be customised in the shop.

What data will be collected?

- Scratch task projects
- Scratch task scores
- Computational Thinking test scores
- Pirate Plunder analytics (information about how they are using the game)
- Written notes on the child's experiences

Sheffield Hallam University | Cultural, Communication and Computing Research Institute

- Audio interviews (which will be transcribed, anonymised and the audio recordings deleted to avoid any chance of identification)
- Questionnaire responses

All data collected during the study will be anonymised. Each child is given an ID number before the study begins that all their data is saved against. The principle investigator then has a paper sheet with the ID number assignments to each child. This sheet will then be destroyed upon conclusion of the study. This anonymous data may be shared with other researchers after the study is complete.

What are the benefits of taking part in this study?
The children get to play a fun and exciting game that should improve their programming and problem-solving skills. The game and other curriculum tasks are in line with the computing national curriculum for KS2. They also get 12 hours of ICT teaching from an industry professional.

What are the risks involved in the study?
The only risk involved in the study is that a child may misinterpret the study as a test of themselves rather than the software. It will be made clear at the start of the study that it is not a test of them but a test of the software and doesn't count towards their school grades in any way.

What are the rights of the participant?
Each child will be told that they can withdraw from the study at any time. Should a child withdraw, or parental consent is not obtained, the child will do a suitable alternative classroom activity.

What if I have concerns about this research?
If you are worried about this research, or if you are concerned about how it is being conducted, you can contact the principal investigator at simon.p.rose@student.shu.ac.uk.