Technical Report

UCAM-CL-TR-890
ISSN 1476-2986

Number 890

Using multiple representations to develop notational expertise in programming

Alistair G. Stead

June 2016

This technical report is based on a dissertation submitted May 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Girton College.

ISSN 1476-2986

Abstract

The development of expertise with notations is an important skill for both creators and users of new technology in the future. In particular, the development of Notational Expertise (NE) is becoming valued in schools, where changes in curricula recommend that students of all ages use a range of programming representations to develop competency that can be used to ultimately create and manipulate abstract text notation. Educational programming environments making use of multiple external representations (MERs) that replicate the transitions occurring in schools within a single system present a promising area of research. They can provide support for scaffolding knowledge using low-abstraction representations, knowledge transfer, and addressing barriers faced during representation transition.

This thesis identifies analogies between the use of notation in mathematics education and programming to construct the Modes of Representational Abstraction (MoRA) framework, which supports the identification of representation transition strategies. These strategies were used to develop an educational programming environment, DrawBridge. Studies of the usage of DrawBridge highlighted the need for assessment mechanisms to measure NE. Empirical evaluation in a range of classrooms found that the MoRA framework provided useful insights and that low-abstraction representations provided a great source of motivation. Comparisons of assessments found that a novel assessment type, Adapted Parsons Problems, produced high student participation while still correlating with code-writing scores. Results strongly suggest that game-like features can encourage the use of abstract notation and increase students' acquisition of NE. Finally, findings show that students in higher year groups benefitted more from using DrawBridge than students in lower year groups.

Acknowledgements

My first thanks go to my supervisor Dr. Alan Blackwell, who has given me the support and freedom to develop my ideas, has maintained my focus by providing an expert balance of encouragement and caution, and has been a constant source of inspiration throughout my Ph.D. My thanks also go to Dr. Andrew Rice, who kindly offered to supervise me during Alan's sabbatical in my first year.

I am grateful to Dr. Sue Sentance and her colleagues in the Computing at School (CAS) network for providing interesting discussions and support for recruitment of the teachers at each of the eleven schools that participated in the interviews and studies presented in this thesis.

This work would not have been possible without financial support from the Engineering and Physical Sciences Research Council, The Computer Laboratory, and Girton College. Special thanks go to Lise Gough, who has always made time to help and support me during my Ph.D. and M.Phil Degree.

Thanks go to members of the Rainbow Group for providing a warm, interesting, and exciting place to work. In particular, I would like to thank my office mates Zhen Bai, Advait Sarkar, Isak Herman and Guo YU for their interesting conversations, endless optimism, help and support over the years.

I owe a tremendous debt of gratitude to my family, all of whom have made significant contributions in support of my education: my mother, Gillian Stead, who has been a constant pillar of love and support throughout my life, my late father Graham Stead, who provided a shining example of what can be done through hard work and commitment, and my sister, Victoria Stead, who has been a continuous source of encouragement.

Finally, I would like to thank my Wife and best friend Clara Stead, who by now knows this thesis almost as well as I do. Without Clara's constant support, endless tolerance and welcome distractions this thesis would not have been possible.

Contents

Glossary

Adapted Parsons Problems (APP) – An extension of Parsons Problems, a type of assessment-type that requires students to reorder lines of programming code to complete a particular goal. Adapted Parsons Problems require the reordering of individual syntax elements in a statement or block of code, page 212.

Computational Thinking (CT) - "*thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out*" (Wing, 2014), page 47.

Computer Science 1 (CS1) – Popular term for the first year undergraduate Computer Science course, page 30.

Computer Science Education (CSEd) – The research area focusing on educational technology, practice and pedagogy within the Computer Science, page 66.

Direct Manipulation (DM) – An interaction mechanism in which the user can control the object of interest through some direct interface, page 85.

Design with Intent (DwI) – a cross-disciplinary framework that emphasises features of design that can be used to coerce users into doing something, page 164.

End-User Programmers (EUP) – Programming to achieve the result of a program primarily for personal, rather than public use (Ko, Blackwell, et al., 2011) , page 38.

External Representation (ER) – a representation not held in the brain, but rather on paper, a computer screen or some other media, page 39.

Integrated Development Environments (IDE) – Programming environments that include support for tools that assist in the development process (e.g. debugging, collaboration, visualisation, distribution), page 19.

Limited Abstraction Representational Systems (LARS) – A representation that can refer to multiple models, page 55.

Minimal Abstraction Representation Systems (MARS) – A system that can refer to exactly one model, page 55.

Multiple Choice Questions (MCQ) – Questions with several choices of which one is correct, page 143.

Multiple External Representations (MER) – Two or more representations not held in the brain, page 20.

Modes of Representational Abstraction (MoRA) – A framework of representation created in Chapter 3 by drawing analogies between Mathematics education and programming, page 20.

Motivation Intervention (MI) – Features created to encourage users to interact with abstract notation (see Chapter 8), page 175.

Notational Expertise (NE) – The development of suitable mental operations to manipulate notations (see Section 3.4.1 for a full definition), page 21.

Parsons Problems – An automated, interactive tool that allows the specification of a program using drag and drop of individual program lines from a set of potential lines, page 146.

Programming by Demonstration (PbD) – A programming system that allows users to specify a program by carrying out actions they would like to automate while the system observes (Cypher, 1991; Smith, Cypher, & Tesler, 2000). Once observation is complete, the system can repeat the actions at will, page 38.

K-12 – Primary and secondary school education in the United States, Canada, South Korea, Turkey, Philippines, and Australia, page 52.

Zone of Proximal Development (ZPD) – An area of learning close to the bounds of one's own area of skill. "Learning awakens a variety of internal development processes that are able to operate only when the child is interacting with people in his environment and in cooperation with his peers. Once these processes are internalised, they become part of the child’s independent development achievement.” (Vygotsky, 1978), page 29.

Unlimited Abstraction Representation Systems (UARS) – A representation that can represent an unlimited number of models, page 55.

Chapter 1 Introduction

1.1 Background

The ability to develop expertise with notations is an important skill that will be essential to both creators and users of new technology in the future. Notations are principally a tool for communication, in which technology user and machine have a shared understanding of the meaning assigned to individual notational elements, the relationship between elements, and how each element is combined to form a whole. Notations are common in music, mathematics, physics, and even chess. However, notations are becoming increasingly used as a medium to communicate with computers.

New notations are being created to support human-computer communication in two ways. First, as has been the case for the last 50 years, new programming languages are being created to allow programmers, who have developed expertise with these formal notations, to improve their application development process. Second, new kinds of notation are being developed to support more "natural" communication with computers. Even modalities such as voice are becoming more notational. Consider a person who uses voice to interact with a smartphone by telling it to create a reminder for 11 o'clock the next day. Despite the convenience of this communication, it is not natural, and requires that the user develop an understanding of the range and structure of input allowed. After many failed attempts, the user learns that particular words or phrases, such as "tomorrow", "PM", "Afternoon" or "Evening" are required to set the right time. If we consider these elements to be parts of a notation, then the user is implicitly developing notational expertise, and using it to program their smartphone to remind them in the future. This thesis explores the characteristics of notational expertise, how a novel kind of programming environment can be used to support its development, and the ways in which successful acquisition of notational expertise can be measured.

The ability to use notational expertise to create computer programs is a skill with great commercial value. However, the high demand for skilled Computer Scientists is not being met; leading technology figures (BBC, 2011) and reports from academic associations such as The ACM and The Royal Society have called for significant improvements in both UK (The Royal Society, 2012) and US (ACM, 2010) national curricula to refocus Computing Education from skill-based competencies, such as using a computer to search for information on the internet, to core Computer Science competencies that do not change over time, such as algorithms, data structures and programming. Recent changes to secondary education curriculum, and the development of complementary computing initiatives such as Google's CS-first Groups (Google, 2015), Code Club (CodeClub, 2015) and Code.org (Code.org, 2015), have attempted to address many of the issues raised in these reports by teaching programming and core Computer Science concepts to students between the age of 4 and 14.

High employability is not the only benefit for students pursuing Computer Science. By creating, managing and debugging complex programs, students acquire notational expertise, which is likely to help to develop and extend problem-solving skills more generally. In his book "Mindstorms", Papert argues that "powerful computation technology and computational ideas can provide children with new possibilities for learning, thinking and growing emotionally as well as cognitively" (Papert, 1980). Learning to program can also be intrinsically motivating, allowing students to express themselves in new ways, and giving insight into how computers work, which is fundamental to our modern, computer-mediated society. Advocates of Computational Thinking suggest that students who acquire the skills to develop computational solutions can apply them to almost any domain (Wing, 2006).

However, learning to program is difficult. When students reach university in the UK, they are more likely than students in any other discipline to drop out of their course, with attrition rates of 22% (Higher Education Funding Council for England (HEFCE), 2013). The USA experiences a similar problem, with attrition rates of 59% - the second highest overall and highest in STEM subjects (National Centre for Education Statistics, 2013). Analysis of student surveys suggest that programming difficulty is a contributing factor to the decision to leave the course (Beaubouef & Mason, 2005; Kinnunen & Malmi, 2007).

In learning to program, students must master both practical skills, including familiarity with programming environments, and expertise with notations used in the environment; and theoretical skills such as programming semantics, algorithm complexity, data structures, and the theory of computation. These skills are difficult to teach, with students requiring high levels of support to overcome several obstacles, including making sense of the large amount of new information presented to them, transferring relevant knowledge from other subject areas, and overcoming practical barriers raised by syntax and semantic programming errors.

Computer Science is a relatively new field that is still developing, with new programming paradigms, languages, and environments being created regularly, and adopted quickly. When creating programming courses, teachers must determine the most suitable language and environment for their students to use in order for them to meet their educational objectives. Teachers must also develop strategies to allow students to transfer knowledge to new technologies in the future. In parallel, technologists must determine the needs of both teachers and students, in order to create educational tools that are appropriate for the needs of the course.

1.2 Research Motivation

Although executable programs can be created by simply editing and compiling text files using a command line interface, almost all programming today is done using Integrated Development Environments (IDEs), which provide users with features to support programming tasks such as integrated debugging, testing, visualisation, collaboration and binary distribution. Improving the usability of these environments for users with little or no programming experience is a major area of research that has resulted in the creation of IDEs developed specifically for such users. These educational IDEs enable the specification of programs using one or more programming notation (e.g. text, visual blocks, diagrams, flow charts), with which users can create, modify, and juxtapose programming logic to be used during program execution.

Blackwell suggests that programming is an activity in which "Interaction with abstractions is mediated by some representational notation, and there are common properties of notations that determine the quality of that interaction." (Blackwell, 2002b). There is also wide agreement that no single programming language is suitable for all tasks (Green & Petre, 1996a; Stenning & Oberlander, 1995; Zhang & Patel, 2006) and that the use of multiple representations may be beneficial to students in education (Ainsworth, 1999). These perspectives, in combination with the development of an analogical framework of representation based on mathematics and computing education, presented in Chapter 2, and further validation from exploratory interviews conducted with teachers, presented in Chapter 4, led to the development of the overall research goal of this thesis, which is to investigate the extent to which multiple representations could be used to develop notational expertise in educational programming environments.

There is little research directly linking theories of representation and multiple representations with the acquisition of programming skill in a classroom environment (see Chapter 2). Changes in the national curriculum, which advise that children as young as four years old should learn to program, suggest there will be increased classroom-based teaching of programming in future, and that it will be taught using many different tools and languages throughout primary and secondary school. These changes suggest that investigation into the use of Multiple External Representations (MERs) in educational programming environments is a fruitful area of research that might help to support students in making the transition between programming representations.

The investigation provides three kinds of benefit to the computer science education and the psychology of programming communities. First, exploration of the research goal is likely to help to assess whether the use of MER programming environments is a promising future direction for the development of educational programming environments.

Second, results from investigations of the research goal can be used to improve the development of single-representation environments by identifying the barriers that students experience when making the transition between representations (i.e. difficulty using new syntax, adapting to new semantics, and adapting to new tools), the improvement of representation transition strategies (i.e. the order of representations to use), and the most appropriate way for students to make the transition between two representations (i.e. the support and type of correspondences required).

Third, investigation of the research goal can increase the validity of comparisons between programming representations, by developing MER tools that use consistent features and semantics. The MER tools could also be used to simulate and improve the way students make the transition between single-representation programming environments.

1.3 Research Questions

The questions addressed by this thesis arise from a conceptual framework referred to as The Modes of Representational Abstraction (MoRA) framework, presented in Chapter 3, which was developed as a result of a review of related literature and tools associated with educational programming research described in Chapter 2. The MoRA framework was developed through analogy between representation use in programming and mathematics education, and was used to identify strategies of transition between programming representations. It was also used to define Notational Expertise (NE), a set of criteria that describe expertise in the use of multiple notations. The development of these ideas led to the overall research goal of this thesis:

To what extent can Multiple External Representation systems (MERs) be used to improve student acquisition of Notational Expertise?

To explore and validate representation transition strategies generated from the MoRA framework, a prototype MER system was created (see Chapter 5) which allows users to view, and make sequential transitions between, representations. Design decisions taken during the creation of the tool, coupled with multiple strategies identified in the MoRA framework, led to the development of several research questions that are investigated in Chapter 6 and used to shape subsequent designs of the system. The first such question is concerned with whether low-abstraction representations improve students' acquisition of NE in MER systems. The second question is concerned with whether the order of symbolic representations (visual block and text) would affect students' ability to acquire NE in MER systems. The third question is concerned with whether the combination of low-abstraction representations and the order of symbolic representations would affect students' ability to acquire NE in MER systems.

The study carried out to investigate these questions generated further questions relating to the use of MERs in educational programming education. The fourth research question, addressed in Chapter 7, investigated the type of assessment that would be most appropriate for measuring student acquisition of NE. Chapter 8 presents an update to DrawBridge, which included new motivation intervention features and integrated assessment features, created to address questions raised in earlier studies. Chapter 8 also presents further studies to examine a fifth research question, which considers whether the addition of motivation intervention features would encourage users to move to symbolic representations and therefore increase acquisition of NE; the sixth research question, which investigated whether the year group of the student affects their ability to acquire NE; and the seventh research question, which investigated the interaction between the year group of the students and the use of motivation intervention features.

#	Question	Addressed in
RQ1	To what extent do low-abstraction representations improve acquisition of NE in MER systems?	Chapter 6
RQ2	To what extent do the order of symbolic representations (Visual-First and Text-First) affect students' ability to acquire Notational Expertise in MER systems?	Chapter 6
RQ3	To what extent does the combination of the use of low-abstraction representations, and the order of Visual and Text representations, change acquisition of Notational Expertise?	Chapter 6
RQ4	What type of assessment would be most appropriate for measuring Notational Expertise?	Chapter 7
RQ5	To what extent can motivation intervention features increase the use of symbolic representations and therefore improve acquisition of Notational Expertise?	Chapter 8
RQ6	To what extent does student year group affect the efficacy of DrawBridge?	Chapter 8
RQ7	To what extent do motivation intervention features affect acquisition of NE in students of different year groups using DrawBridge?	Chapter 8

1.4 Thesis Overview

Chapter 2: Literature Review

Chapter 2 presents a literature review of areas of research relevant to the development of programming competence, including cognitive psychology, education theory, and the psychology of programming. A review of theories of representation is also presented, together with a review of educational programming systems categorised by their representation. Finally, a review of systems using Multiple External Representations (MER) is presented, which provided initial motivation to investigate the research goal of using MER systems to improve educational programming environments.

Chapter 3: Developing Notational Expertise in Programming

Chapter 3 extends discussion of theories of representation presented in Chapter 2 by drawing analogies between programming and mathematics. A review of mathematical literature concerned with representation, and a comparison between two campaigning movements, The New Math of the 1960s and Computational Thinking, are used to develop an analogical framework, which classifies programming representations by their Modes of Representational Abstraction (MoRA). The MoRA framework enables the identification of representation transition strategies that minimise the barriers between low-abstraction, educationally suitable programming representations, and high-abstraction professional representations. A set of success criteria, referred to as Notational Expertise (NE), is defined to identify programmers who are able to successfully make the transition to use programming notations. Finally, a discussion of the way in which strategies identified using the MoRA framework can be used to guide design heuristics of educational MER programming tools is presented, which forms the primary research goal of whether MER systems can be used to help students to make transitions and identify correspondences between representations, and therefore acquire NE.

Chapter 4: Teaching Interviews

Chapter 4 describes the investigation of current teaching practice in schools via a series of semi-structured interviews conducted with secondary school teachers and professional software developers who have experience of running after-school programming clubs. Thematic analysis is used to identify common teaching strategies, decisions affecting the choice of educational programming tool, and problems encountered by students learning to program in a classroom environment. Finally, interviews are used to elicit expert feedback of an early prototype of a MER educational programming environment.

Chapter 5: DrawBridge

Chapter 5 presents design decisions and implementation constraints encountered during the creation of DrawBridge, a configurable MER educational programming environment that allows students to view pairs of programming representations and make transitions between them using strategies from the MoRA framework. The system is used to investigate the acquisition of NE with MER systems in studies presented in Chapter 6 and 8.

Chapter 6: Manipulation of MERs to Encourage Notational Expertise

Chapter 6 presents two quasi-experimental studies that compare four configurations of programming representations in DrawBridge, which are distinguishable by two factors: the order of symbolic representations (visual blocks before text, and text before visual blocks), and the inclusion of concrete representations. The studies, created to investigate design decisions made in DrawBridge, confirm that students using visual representations before text representations improve more than those using text representations first, and that low-abstraction representations provide major motivation for using MER systems. The final experiment – a follow up think-aloud study to investigate pre-test differences in the first study, led to the identification of limitations in the assessment of NE, which is addressed in Chapter 7.

Chapter 7: Designing Assessments for Notational Expertise

Chapter 7 responds to limitations of the assessments identified in the previous chapter, and calls for assessment in educational programming environment to be more rigorous, by presenting a study to compare four candidate assessment types for measuring NE: code writing, multiple-choice, debugging and a Adapted Parsons Problems, in order to answer a further research question: what type of assessment would be most appropriate for measuring Notational Expertise?

Chapter 8: Motivation Intervention to enhance Notational Expertise

Chapter 8 presents a two-part study to investigate the effect of new motivation intervention features, developed in version 2 of DrawBridge, on students' acquisition of NE. Motivation intervention features were developed as a response to instrumentation results collected during the study reported in Chapter 6, which suggested that low-abstraction representations might be too engaging, and distract students from making the transition to more abstract, symbolic representations. The study used an integrated version of Adapted Parson Problems, which was identified as the assessment that provides the most appropriate measure of NE. Results from the study strongly suggested that motivation intervention features do increase acquisition of NE, but that an increase in time viewing symbolic representations is not the only factor responsible for improvement. Results also suggest that student year group is a significant factor in their acquisition of NE.

Chapter 9: Conclusion

Chapter 9 presents the major findings in response to the overall research goal of this thesis, and response to each of the contributing research questions, followed by their implications for the development and assessment of NE and educational programming environments. Finally, the implications of this research and directions of future research are discussed.

1.5 Relevant Publications

1. *Stead, A. (2014, August). How Can We Improve Notational Expertise? In Proceedings of the Tenth Annual Conference on International Computing Education Research, (pp. 173-174). ACM.*

2. *Stead, A., & Blackwell, A. (2014, June). Learning Syntax as Notational Expertise when using DrawBridge. In Proceedings of the Psychology of Programming Interest Group (pp, 41-52).*

3. *Stead, A. G., & Blackwell, A. F. (2013, July). Representational Formality in Computational Thinking. In Proceedings of the Psychology of Programming Work in Progress (pp, 27-29).*

Chapter 2 Literature Review

This chapter contains a review of literature in principal areas of research surrounding the development of educational programming environments, including cognitive psychology, education theory, and the psychology of programming.

The structure of this chapter is as follows: Section 2.1 reviews three influential perspectives of child cognitive development, which provide a basis for a review of educational theory that has influenced programming teaching practice. It describes the theory of knowledge transfer, and how this is applicable in educational programming. Section 2.2 reviews changes to computing curricula that have influenced the teaching of programming in schools. It also discusses individual differences in learners that affect how educational programming tools are used. Section 2.3 describes existing educational programming environments categorised by their use of representation, and the suitability of each type of representation to novice programmers. Section 2.4 describes the types of representation that exist and their suitability to different tasks. Further, it discusses multiple representations, how they have been used in educational programming, and studies relating to their effectiveness.

2.1 Cognitive Perspectives and Learning theory in Programming Education

Research and development of educational programming environments has been influenced by three main perspectives of child cognitive development related to constructionism – a theory of learning that has influenced programming education research and the development of educational programming environments over the last 50 years. Each perspective describes a model for the way in which children build and process knowledge, and provides a basis for reviewing educational theory and the use of programming tools in computing education. These theories also provide the foundation for the MoRA framework of representation, developed in Chapter 3, and raise the question of how cognitive development affects acquisition of notational expertise using multiple representations, explored in Chapter 8. Furthermore, this section reviews theories of learning that have also directly influenced teaching practice of programming, such as scaffolding and cognitive apprenticeship, which describe support given to assist learners in carrying out tasks they would not otherwise be able to complete. Finally, the section describes the theory of knowledge transfer, the types of transfer that are possible, how students can use transferred programming knowledge, and the influence transfer has on the design of educational programming systems.

2.1.1 Child Cognitive Development

One of the most influential educational theories in the development of programming environments has been Papert's theory of *constructionism*. In his famous book Mindstorms, Papert states that children can learn concepts by using physical and digital "objects-to-think-with", such as gears, or the turtle in his LOGO system. Papert argues that such objects can be an effective tool for learning as they are already embedded in the culture around a child, and the child can use knowledge of his own body to imagine how the object can move (Papert, 1980). Papert's work was highly influential in this thesis, motivating the use of hand-drawn characters (objects-to-think-with) as a starting point for programming in Chapter 5.

Papert's theory was influenced by Piaget's theory of *constructivism*, which states that children achieve intellectual growth via the use or adaptation of organised patterns of thought, or schemas. In this theory, children are able to *assimilate* new information by using existing schemas, and *accommodate* new information by creating new schemas. Piaget's experiments showed that child cognitive development occurs in four distinct, sequential stages: sensorimotor, preoperational, concrete operational and formal operational. Each stage is only reached after a surge in development from the previous stage, and occurs according to biological maturation. The use of programming languages typically requires logical or operational thought. Piaget states that this appears before adolescence in the third stage, "concrete operational", where the child can think logically, but can only apply their knowledge to physical concrete objects. In order to manipulate symbols and think logically without the use of concrete objects, Piaget states the child must reach the "formal operational" stage, which occurs during adolescence. From Piaget's perspective of constructivism, a child's developmental stage is likely to influence their ability to learn to program, particularly with conventional text programming tools that require the user to manipulate symbols and think in an abstract manner. Piaget's theory has influenced curriculum design (Halsey & Sylva, 1987) and has been used as a template for computing education researchers for assessment marking (Gluga, Kay, Lister, & Teague, 2012) and data interpretation (Teague, Corney, Ahadi, & Lister, 2013).

Bruner provides an alternative perspective of constructivism, which is drawn on in later chapters of this thesis. He states that instead of developing in four distinct stages, child cognitive development is "…more like a staircase, with short risers. More a matter of spurts and rests" (Bruner, 1968). Bruner argued that humans have three parallel systems for processing and representing things: **Enactive** – for manipulation or actions for which we don't have words or symbols; **Iconic** – for perceptual organisation and imagery; and **Symbolic** – for language. He stated that these processing systems were loosely sequential, and that the ability to use symbolic processing developed last, at around age 7. This perspective of learning suggests that some representations may not be suitable as first programming languages, and that there might be an optimal order in which to introduce programming representations to children. In fact, Bruner stated that for both child and adult learners, moving from enactive, to iconic, to symbolic representations is effective when faced with new material (Bruner, 1968). When Alan Kay outlined his vision for a personal computer for children of all ages (Kay, 1972), he supported the use of Bruner's three modes of representation in determining the educational goals that would be feasible at various ages. Kay was also influenced by Bruner when developing the first prototypes of the graphical user interface at Xerox Park:

> *"The work of Papert convinced me that whatever user interface design might be, it was solidly intertwined with learning. Bruner convinced me that learning takes place best environmentally and roughly in stage order – it is best to learn something kinaesthetically, then iconically, and finally the intuitive knowledge will be in place that will allow more powerful but less vivid symbolic processes to work at their strongest." – Alan Kay* (Kay, 1991)

A third perspective on constructivism is that of the social constructivism present in Vygotsky's work. Vygotsky rejected the idea that development always preceded learning, and stated that by learning within the Zone of Proximal Development (ZPD) – the area between what is known and what is not known, children could learn skills beyond their current stage of development when given assistance by a more capable person (Vygotsky, 1978).

Although constructivists argue that children create their own knowledge, many students face difficulties mastering complex ideas or tasks. One teaching approach adapted from Vygotsky's ZPD, is to provide a child with the support required to complete a task that would normally be beyond their capability when unassisted. This approach, known as scaffolding, asserts that the learner is able to concentrate on the parts of the task they are able to complete, while increasing the speed at which they develop competence in parts they cannot complete (Wood, Bruner, & Ross, 1976). This approach is used to determine the order of representations during the design of DrawBridge in Chapter 5. Cognitive apprenticeship, an extension of scaffolding that focuses on acquiring cognitive and metacognitive skill through guided experience (A. Collins, Brown, & Newman, 1989), has been successfully used as a method of teaching Computer Science 1 (CS1) undergraduate lectures (Cutts, Esper, Fecho, Foster, & Simon, 2012) and has helped to reduce dropout rates in CS1 courses (Vihavainen, Paksula, Luukkainen, & Kurhila, 2011).

2.1.2 Designing Educational Programming Environments for Transfer

Recent changes in curricula, discussed in the next section, which encourage students as young as four years old to start learning to program suggest that students will need to transfer knowledge between multiple programming languages and environments. Perkins and Martin argue that learners may not possess effective strategies to protect new knowledge, and that new knowledge created by novice programmers is “fragile”, usually partial, hard to access and often misused (Perkins & Martin, 1986). They also state that novice programmers may be unable to recall “inert” knowledge, which is knowledge they acquired in a particular context, but find difficult to apply or relate to a new context. Inert knowledge can be identified if giving the learner hints triggers success. Accessing inert knowledge is a problem of transfer.

What Kinds of Transfer are Possible?

In order to exploit existing representational knowledge when learning to use new programming representations, students must be able to transfer understanding gained in one context so it can be applied in another. Perkins and Salomon define two methods of achieving transfer (Perkins & Salomon, 1988): low-road transfer, which occurs when a new context and old context are sufficiently similar that knowledge can be applied directly to the new context with minimal alteration (for example, a guitarist choosing to play the ukulele may be able to transfer fingering and strumming techniques, but may have to adapt to the different string tuning); and high-road transfer, which occurs when key characteristics are abstracted from an old context and applied to a new context, or characteristics are abstracted from a new context to search for previous examples. Perkins and Salomon also define two methods for teaching for transfer; *hugging* is the facilitation of low-road transfer, and occurs when teachers adapt the teaching context to be as similar as possible to the application context. *Bridging* is the facilitation of high-road transfer, and occurs when students are taught to identify abstract properties of the method or object of interest, generalise those properties, and apply them to new domains. The concept of Bridging is central to development of DrawBridge – a prototype development tool to support transfer between multiple representations, presented in Chapter 5.

Can Educational Programming Environments Support Transfer?

Teaching for transfer is often cited as an important design goal for developing recent educational programming systems. In novice programming[1], Pane and Myers encourage designers of programming systems to maximise transfer by choosing appropriate metaphors, ensuring consistency of metaphors and increasing consistency with external knowledge (Pane & Myers, 1996). Many recent computing education systems provide mechanisms for transfer: Alice 2 uses both hugging and bridging approaches to encourage students to transfer knowledge to more general-purpose languages. For example, block text can be changed to reflect Java syntax (hugging), and projects can be exported to be used in professional environments such as Eclipse (bridging) (Kelleher, Cosgrove, & Culyba, 2002). SmallBASIC also encourages students to "graduate" by exporting their projects to Visual Studio, which automatically converts code to Visual Basic (Microsoft, 2008).

How Can Teachers Support Transfer?

Teachers are able to facilitate transfer if appropriate tools and techniques are used. For example, explicit bridging techniques such as analogy and generalisation have been shown to result in significant improvements in exam scores when teaching for transfer in undergraduate programming courses (Dann, Cosgrove, Slater, & Culyba, 2012). Hundhausen found that in order to promote positive transfer to text-based representations, one could use Direct Manipulation tools to provide a "way in" to traditional text programming (Hundhausen, Farley, & Brown, 2006).

Despite the evidence to suggest transfer methods can have a positive impact on student learning, more general studies evaluating transfer methods have shown mixed results, and have often been reported using different performance measures, task memory demands and number of hints (Barnett & Ceci, 2002). Perkins and Martin posit that without conditions conducive to low-road or high-road transfer, programming knowledge becomes inert (Perkins & Martin, 1986). Inert knowledge can be identified when participants achieve success when given hints or suggestions. Examples of instances of poor transfer include a study by Parsons and Haden, which found minimal transfer for students attempting transfer from visual programming environments to text-based environments, where students struggled to make any connection with the visual programming and text programming (Parsons & Haden, 2007).

[1] The term "novice" is sometimes used as a proxy for "all users who are not expert programmers".

Powers also found that students continued to experience "syntax overload" when moving from visual programming to text languages, but that problems were worse when the distance of transfer increased (Powers, Ecott, & Hirshfield, 2007). The problem of inert knowledge and difficulty of transfer is not just experienced by novice programmers; even experienced programmers find bridging programming concepts to new representations difficult (Scholtz & Wiedenbeck, 1990).

2.2 Computing Education in the Classroom

There are practical concerns that have a direct effect on student success and pedagogical strategies used to teach programming in schools. These concerns affect the way in which educational programming environments are used in the classroom, the goals of teachers and students using them, and consequently the design and assessment of the tools presented in this thesis.

2.2.1 Curriculum and Pedagogy

The content of computing lessons in schools is greatly influenced by national curriculum and exam syllabuses. From the early 1990s, students in the United Kingdom studied Information Communication Technology (ICT), which contained minimal amounts of programming content, and instead emphasised word processing, spreadsheet, presentation and desktop publishing skills (The Independent ICT in Schools Commission, 1997).

The new National Curriculum intends to increase the rigour of computing and encourage more students to become software engineers (Wells, 2012). It states that students should learn core computing concepts, Computational Thinking abilities (Wing, 2006), and two programming languages, at least one of which is text-based (DfE, 2013). Interviews conducted with teachers regarding current practice, presented in Chapter 4, suggest that a combination of pedagogical approaches are currently being used, including the "syntax-free" approach, in which teachers are using visual block languages such as Scratch to teach practice-based skills of programming, and then the transition to a "literacy" approach, in which students begin using a "real" language such as Python or JavaScript (Fincher, 1999).

Pedagogical approaches that aim to improve instruction by increasing motivation and encouraging student interaction have achieved promising levels of success. For example, Peer Instruction techniques, which encourage students to work in pairs have been shown to successfully improve results, allowing students to help each other solve problems and take on different roles in order to deal with high and low-level concerns (Simon, Kohanfars, Lee, Tamayo, & Cutts, 2010). Although typically used in professional programming practice, studies of undergraduate Pair-programming show promising results, with students completed assignments more quickly and to a higher quality (Williams & Upchurch, 2001).

2.2.2 Perceived Self-Efficacy

When designing educational programming tools, it is important to consider how student participation can be encouraged and how the tools can be developed to support students of different ability levels, particularly if low participation is due to lack of support for low-ability students. Self-efficacy, or the justified level of belief in one's own capability (Bandura, 1997), can affect student success when using educational tools. Self-efficacy can be developed from many sources, including through "mastery" experiences, in which people develop a sense of their own ability; through vicarious experience, in which people see others they consider similar to themselves succeed; and through social persuasion. Students with low self-efficacy are likely to expend less effort on a task, and may exhibit reduced coping behaviour when task completion becomes difficult (Bandura, 1977). Experiments investigating the differences between males and females completing end-user debugging tasks found that female self-efficacy fell when using environments with high-support, and that lower self-efficacy in females was predictive of the amount of experimentation or "tinkering" they carried out, which was found to be significantly tied to increased understanding, and successful testing and debugging (Beckwith et al., 2006). Ryan showed that students with low self-efficacy would benefit from additional support from teachers, fellow students, or software tools, as they are less likely to ask for help in the classroom (A. M. Ryan, Gheen, & Midgley, 1998).

Gender Differences

When designing educational programming environments, it is important to identify features that may be gender biased. There is evidence to suggest that there is an increasing gender gap in secondary school computing education. Margolis and Fisher attribute this gap to an "all-boy club" computing room culture, pedagogy design that primarily motivates males, and loss of confidence for females during adolescence (Margolis & Fisher, 2003). The gap may also be affected by students' self-efficacy, which Busch shows is lower for females when regarding the completion of complex tasks (Busch, 1995). Beckwith and Kissinger show that there are differences in the level of tinkering between males and females, with males more likely to tinker than females. The level of tinkering however, does not necessarily lead to improved understanding, as it can be counterproductive, reducing time available for reflection, which improves efficacy (Beckwith et al., 2006).

One suggested method of reducing the gender differences is to accept the validity of multiple ways of knowing and to "give equal access to elements of computation" (Turkle & Papert, 1992). This view states that instead of only supporting the dominant "hard" thinking style, which is preferred by males and encourages abstract thinking and systematic planning, educators should also support a "soft" thinking style, which encourages closeness to objects and supports concrete reasoning. Courses such as "Media Computation" have been successful in attracting and retaining a large proportion of female college students by teaching core computer science concepts rooted in the manipulation of media (Guzdial, 2003).

Gender-specific improvements to existing educational programming systems have shown that it is possible to design tools that better support female students' goals and learning behaviours. For example, Kelleher carried out studies using a specialised storytelling version of the Alice programming system that resulted in increased female student motivation and increased interest in enrolling in future programming courses (Kelleher, 2006).

2.3 Educational Programming Environments

The usability and design of programming environments for novices has been a major focus of research for more than 40 years (Pane & Myers, 1996). Kelleher's taxonomy of novice programming environments shows that attempts have been made to make general purpose languages more understandable, reduce error proneness in system interaction, and tightly integrate required features for novice users (Kelleher & Pausch, 2005). In addition to improvement of existing techniques, investigations into the use of non-conventional programming techniques, such as Direct Manipulation and Programming by Demonstration, have provided alternative ways to reduce barriers for novices.

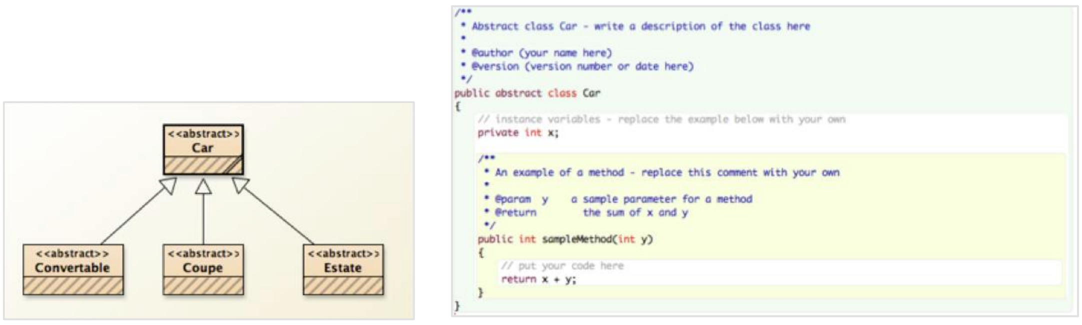
2.3.1 Text-Based Development Environments

Conventional text-based programming environments have experienced modest, iterative improvements since the 1970s to include new usability features such as code completion, integrated compilation and library support. Although these features provide increased support for programmers, novices still find their default environments difficult to use (Rigby & Thompson, 2005).

Novice programmers encounter many difficulties when learning to program (du Boulay, 1986), including *orientation* – finding out what programming is for, *the notional machine* – finding out what the properties of the machine are, *notation* – mastering syntax and underlying semantics, and difficulties learning common structures or plans. Studies have shown that novices pay more attention to syntactic information rather than semantic, which grows with expertise (Adelson, 1981). Recent novice education environments, such as BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003), emphasise learning goals such as the acquisition of object oriented programming concepts, and provide users with highly supportive features such as block highlighting, visualised class structure, and object inspection (Figure 2.1). Despite this support, designers of text-based educational programming environments cannot easily reduce the error proneness of the notation. Data collection of common errors in educational programming tools show that syntax errors such as missing semi-colons or unbalanced brackets account for the majority of errors received by novice users (Brown & Altadmri, 2014; Jadud, 2005).

The high error proneness of text notation is challenging for novice programmers, potentially forming barriers that prevent students from progressing, reducing the likelihood of long-term success. Syntax errors are particularly challenging for novice programmers who may have low confidence levels, and do not yet have effective strategies for identifying syntax issues quickly. Denny et al. showed that students of all levels struggled with syntax errors, with some errors taking the same amount of time for the highest performing students to debug as any other student (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011). Denny et al. also found that weaker students encountered syntax errors they were sometimes unable to fix, halting their progress. These issues, if not supported, could lead to low levels of self-efficacy, particularly if a sense of efficacy is not yet firmly established (Bandura, 1997).

Novices themselves have reported that they still experience difficulty constructing programs, learning syntax and finding bugs in conventional text-based programming environments (Lahtinen, Ala-Mutka, & Järvinen, 2005) (Brown & Altadmri, 2014).



```
/**
 * Abstract class Car - write a description of the class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public abstract class Car
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y   a sample parameter for a method
     * @return       the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}
```

Figure 2.1: BlueJ Java Development Environment

(**Left**: BlueJ UML view showing program class structure, **Right**: BlueJ text view showing block highlighting)

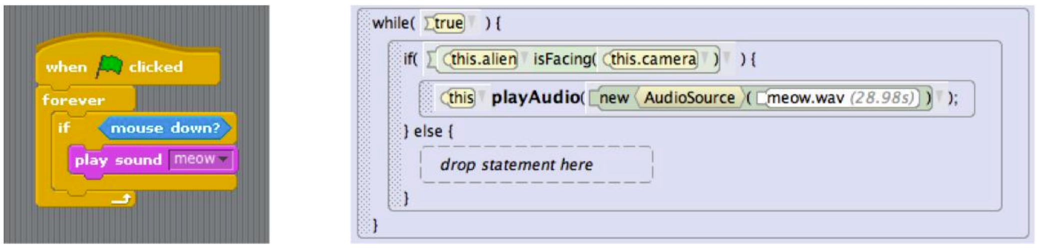
In order to increase student motivation and engagement in programming tools, some educational programming environments provide graphical output alongside traditional text notation (Esper, Wood, Foster, Lerner, & Griswold, 2014; Kolling, 2010; Lee, Ko, & Kwan, 2013; Microsoft, 2008). These interfaces allow students to view the execution of their code and its effect on physical or digital objects, which act as one of Papert's "object-to-think-with". In addition to using on-screen graphics to visualise program execution, both physical (McGill, 2012), and simulated robots (Major, Kyriacou, & Brereton, 2011) have been found to increase student motivation using text-based environments. Systems such as SonicPi, which allows novices to specify synthesized music via a text-based programming language, Ruby, have also been found to increase student motivation and improve recall rates of commands relative to standard environments (Sinclair, 2014).

2.3.2 Visual Language Development Environments

Visual programming languages (VPLs) are languages that allow one to specify a program in two or more dimensions (Myers, 1990)[1]. VPLs provide an alternative notation to conventional text-based notation, and are able to offer several benefits to students in computing education. The primary benefit of VPLs is that they are able to enforce syntactic validity, which reduces error proneness in the notation by making it impossible for novices to encounter syntax errors. An additional benefit of VPLs is that the number of possible statements, keywords and inputs can be limited in order to simplify the environment and thereby reduce learning requirements. Designers of VPLs state that these benefits contribute to a "low-floor" for users, where novices can easily get started, and that the large number of programming tasks possible with VPLs results in a "high ceiling", where novices can iterate to create complex projects over time (Resnick et al., 2009).

[1] While this definition is fairly unsophisticated, and making distinctions between visual and text representations is a complex matter (Shimojima, 1999), it provides a clear, convenient distinction between representations that will be used in this chapter

VPLs have had a great deal of popularity in computing education over the last decade. In addition to providing notational benefits that are more suitable than text code for novices, environments such as Scratch (Resnick et al., 2009) and Alice (Kelleher et al., 2002) provide motivation to students by allowing them to construct games and storytelling scenarios. The tools are also supported by comprehensive teaching resources, user documentation, and online communities in which projects can be shared, discussed and downloaded.



```
when 🏁 clicked
forever
  if mouse down?
    play sound meow
```

```
while( true ){
  if( this.alien isFacing( this.camera ) ){
    this playAudio( new AudioSource( meow.wav (28.98s) ) );
  } else {
    drop statement here
  }
}
```

Figure 2.2: Examples of Educational Visual Programming Languages

(**Left**: A short program in scratch that plays a sound when the mouse is clicked, **Right**: A similar program in Alice 3)

There is evidence to suggest that VPLs have been successfully used both in and out of the classroom. Meerbaum-Salant showed that, using appropriate learning materials and teaching methods, Scratch could be used within the classroom to develop understanding of computer science concepts (Meerbaum-Salant, 2010). However, responses from teachers during semi-structured interviews presented in Chapter 4 suggest that the easy availability of blocks in Scratch may reduce the need for students to remember the required keywords or syntax. Several researchers showed that Scratch and Alice could be used outside of the classroom to successfully encourage exploratory learning, improve engagement, and increase intrinsic motivation (Franklin et al., 2013; Kelleher, 2006; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). Mishra also showed that VPLs such as Scratch could be used to support CS1 students when making the transition to using text-based languages (Mishra, 2014).

Despite their popularity in computing education, it is well understood that VPLs, like any notation, have limitations (Green & Petre, 1996b). For example, a usability analysis using the Cognitive Dimensions (CDs) of Notations might show that they are likely to have increased *diffuseness*$^{(CD)}$ and reduced *visibility*$^{(CD)}$ dimensions when compared to text notations. The user may also need to spend longer gathering information from the notation when compared with a text-based language (Green & Petre, 1992). In addition to notational issues, studies suggest that some VPL environments such as Scratch may encourage students to develop bad habits such as disregarding initial design or analysis, and avoiding the use of complex control structures (Meerbaum-Salant, Armoni, & Ben-Ari, 2011). It is also clear that one VPL does not fit the needs of all students, and that more restricted, and less text-heavy VPLs may be needed to suit young students, who do not find it easy to parse text-heavy VPL environments such as Scratch and Alice (Flannery et al., 2013; MacLaurin, 2011).

2.3.3 Low-Barrier Educational Environments

A primary benefit of VPLs such as Scratch is that they can enforce syntactic validity by defining the shape of blocks to be such that they can only fit together in a valid way. Structured editors, such as those created using Citrus (Ko & Myers, 2005), similarly aim to limit the error proneness of programming environments by constraining user input to a fixed structure that reflects the underlying abstract syntax tree. These editors have traditionally been text-based, and are not commonly used in programming education. However, research using tangible structured editors such as Perlman's TORTIS Slot Machine shows that children as young as 4 years old can benefit from using structured editors, which can eventually be used make the transition to text languages like LOGO (Perlman, 1976). More recent prototypes of educational programming systems suggest that notation in structured editors may be easier to change than existing VPLs (McKay, 2012).

Other efforts to minimise the barriers to programming have resulted in systems that allow users to bypass the need to interact with a representation to specify the program. Programming by Demonstration (PbD) systems allow users to specify a program by carrying out actions they would like to automate while the system observes (Cypher, 1991; Smith et al., 2000). Once the actions are observed, the program to repeat them is generated and stored to allow the user to re-execute their actions in the future. PbD primarily benefits End-User Programmers (EUPs) – non-professional programmers who use programming to complete a task (Ko, Myers, et al., 2011). PbD on its own is likely to be unsuitable for helping students to learn to program as they would not be required to manipulate representations. However, PbD systems may prove to be useful in programming education systems if they were used to scaffold student understanding by generating programming code based on observed student actions.

An additional way of lowering the barriers to programming is to reify abstract programming code by encouraging students to embody the program and act out each of its command. The CSUnplugged website, for example, provides a set of paper-based classroom resources that allow children to act out programming tasks in order to understand how algorithms work (Bell, Alexander, Freeman, & Grimley, 2008). Surveys given after using CSUnplugged suggest that these resources may act as useful tools to change students' perceptions of computer science (Taub, Ben-Ari, & Armoni, 2009). The materials may also engage students who respond well to the "soft" style of learning, described by Papert and Turkle (Papert & Turkle, 1992), and increase understanding of abstract computing concepts by allowing students to first access the concepts through concrete examples. The Hank tool is another system that allowed students to act out program execution (Mulholland & Watt, 1998). It contained a cognitive modelling language, designed for cognitive psychologists with no programming experience. The processing unit, named Fido, was responsible for program execution. As students could not be required to have a computer at home, they were expected replace Fido, and execute their programs from paper in the same way.

Finally, the ToonTalk system provides an example of a programming environment that does not fit easily into classifications presented above. ToonTalk maps abstract computational concepts to concrete metaphors in a 3D video game world (Morgado & Kahn, 2008). These metaphors are used to link computational ideas to concepts children already understand. For example, birds carrying messages between houses are used to teach the ideas of concurrency and message passing.

2.4 Representational Support in Programming Education

The previous section shows that a large number of educational programming systems can be categorised by the type of representation used. This section further examines programming representations by investigating the benefits of external representations, how humans process them to infer information, and how suitable they are for different tasks.

2.4.1 External Representations

An external representation (ER) can be defined as a representation not held in the brain, but rather on paper, a computer screen or some other media. Larkin and Simon distinguish between two kinds of ER: *sentential*, which is a data structure that consists of a sequence of elements, and *diagrammatic*, which is a data structure containing information indexed by two-dimensional location (Larkin & Simon, 1987). They also state that two ERs are similar if they contain the same amount of information, and require a similar time and effort for humans to compute them. Different ERs can be more or less useful depending on their structure; for example, diagrams can exploit perceptual processes for grouping, which improves recognition and search, while tables improve speed of read-off and emphasise empty cells.

Programming notation can be considered to be any formal representation that can be used to specify a program (Blackwell, 2002b). Green and Gilmore note in their match-mismatch conjecture that the difficulty of the mental operations required to complete certain tasks depends on the suitability of the notation used (Gilmore & Green, 1984). Green also notes in the Cognitive Dimensions (CDs) of Notations framework, that creating new notations requires the designer to make a series of trade-offs that affect the suitability of the notation for a particular task (Green & Petre, 1996b). The suitability of a representation is also likely to be affected by its limits of abstraction. Stenning and Oberlander describe three levels of abstraction in representational systems: *minimal abstraction*, *limited abstraction* and *unlimited abstraction*. They suggest that when compared to linguistic representations such as text, graphical representations such as diagrams can help to process information due to their limits of abstraction (Stenning & Oberlander, 1995).

2.4.2 Multiple External Representations

Improved understanding of the value of suitable representations has led researchers to investigate whether the use of Multiple External Representations (MERs) can improve learner understanding and the ability to complete a task. One commonly identified benefit of MERs is that when encountering a representation unsuitable for the current task, users are able to choose more appropriate representations to enhance their performance (Ainsworth & Van Labeke, 2002; Parnafes & Disessa, 2004).

However, the usefulness of MER systems is thought to largely depend on the suitability of the design of the system. In DeFT (Design, Functions and Tasks), a conceptual framework for understanding the effectiveness of MERs in learning environments (Ainsworth, 1999), Ainsworth identifies several design dimensions for effective MERs, including the number of representations, the information present and its distribution over each representation, the form and sequence of representations, and translations between them.

When considering representations that differ in information content, it follows that together, two or more representations must provide more information than just using a constituent ER. Ainsworth refers to this as the *complementary* role of MERs in DeFT. In addition to complementary roles, Ainsworth states that MERs can serve to *constrain* interpretation by using one representation to limit the interpretation of another. For example, a text description 'coins with a total value of £1' next to a picture showing a 50p coin, two 20p coins and a 10p coin, makes it clear which coins are available, and constrains the number of possibilities described by the text representation. Ainsworth's final function type of MERs in education is to *construct* deeper understanding by allowing learners to create references that expose the underlying conceptual framework.

In the psychology of programming, the CDs framework recognises that creating new notations requires the designer to make a series of trade-offs which result in strengths and weaknesses in the notation (Green & Petre, 1996b). It follows that two different notations would have different strengths, and together, could be more useful than a single notation. Learners using multiple representations would therefore benefit in exploiting MERs by switching between representations to use the most suitable representation for the task they were trying to complete.

Studies using MERs

There are two broad classes of study using MERs. The first includes those where learners construct their own representations. For example, Cox and Brna investigated the creation of ERs in analytical reasoning problems and concluded that there are four stages of reasoning using ERs: problem comprehension, ER selection, ER construction, and ease of use of the subsequent ERs (Cox & Brna, 1995). They also noted that ER selection is difficult because the ER must be capable of supporting the semantics of the problem. It is likely that ER selection could be especially difficult for novices, who do not yet have the expert knowledge required to select the most appropriate ER.

The second class of MER study includes those where learners are presented with multiple representations from the start. Mayer and Sims found that inexperienced students who lacked domain specific knowledge benefitted from both visual and verbal explanations when learning about the human respiratory system (Mayer & Sims, 1994).

Ainsworth showed that the COPPERS system, which used informationally equivalent representations to provide computationally dissimilar descriptions of coins, successfully taught students to consider multiple solutions to coin problems (Ainsworth, 1997). Ainsworth also observed that learning outcomes were improved for students using similar representations compared with students using dissimilar representations in her CENTS system, which used a mixture of eight representations to teach children strategies for problem estimation.

Although MERs have been shown to provide benefits to learners, they do not *always* help learners. For example, Cox shows (Cox, 1996) that learners can switch between representations too often when coming up against a problem; *impasse-driven* switching, also referred to as “thrashing”, is triggered by learner uncertainty of how to proceed, and is associated with poor performance. However, if the learner switches at the correct time, which cox refers to as *task-driven* switching, they can achieve higher performance.

2.4.3 Existing MER Systems

A number of educational programming systems have been created to investigate the benefits of multiple representations. These systems differ by their number of representations used, whether representations are interactive or not, and the types of representation used.

Representations of Programs and their Output

Educational programming systems often provide an integrated user experience to improve usability for novice users. As discussed, several systems allow the user to execute a program and view the execution output in the same window. Although the execution representation is usually only available during or after execution, it offers an additional representation that the novice can use to gather information about the execution of their program. The Jeliot 2000 system (Figure 2.3) allowed users to view a dynamic graphical representation of program execution. A study using the system found that users who had both the programming and execution representations improved significantly, while those with just the programming representation did not (Levy, Ben-Ari, & Uronen, 2003). Other systems, such as ConMan, provide intermediate execution representations at each stage of a data-flow program, allowing users to view the program’s output as they constructed it (Haeberli, 1988). Although viewing execution representations may be useful, if they cannot be modified or interacted with to change the underlying programming structure, they are not able to provide the full power of MERs.

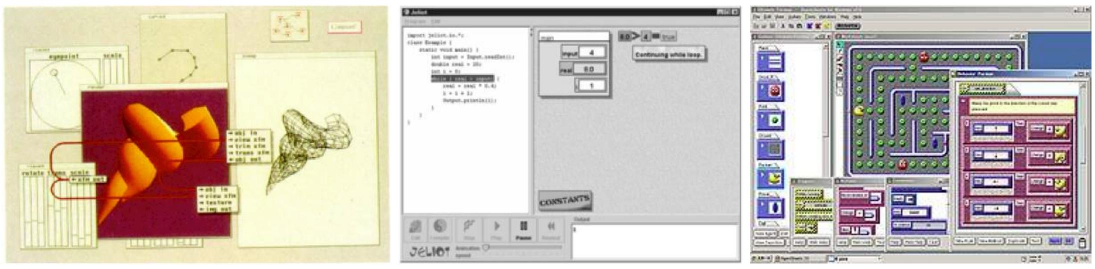
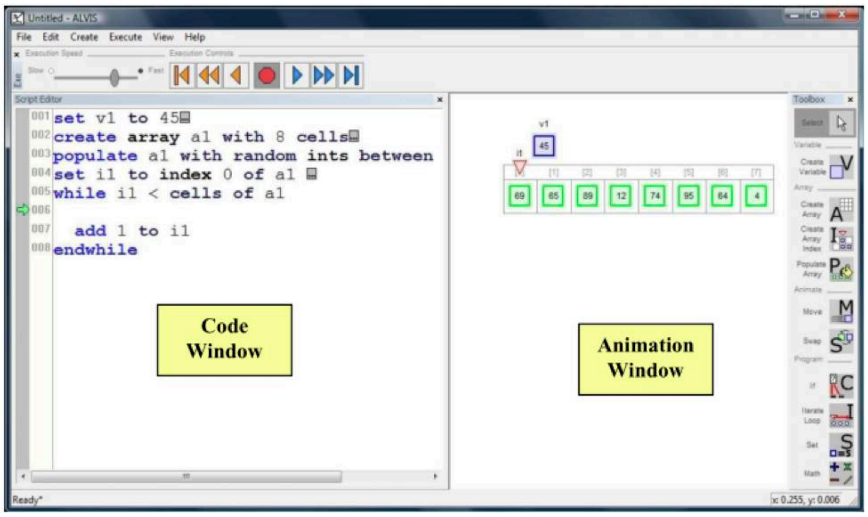


Figure 2.3: Program Visualisation Systems

(**Left**, ConMan, **Middle**: Jeliot 2000, **Right**: AgentSheets)

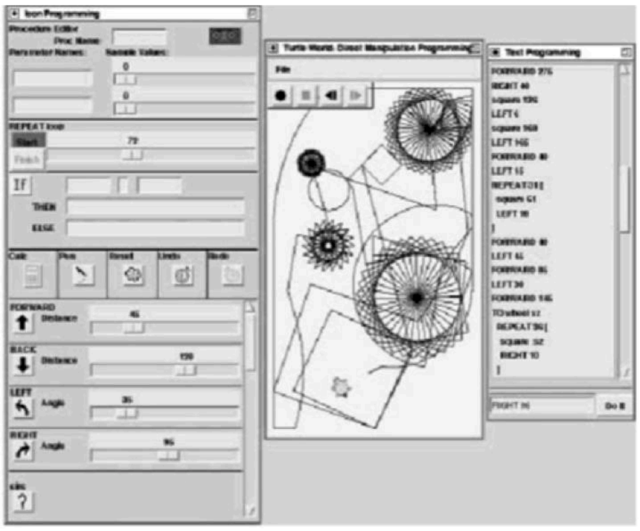
Multiple Interactive External Representations

Some computing education systems allow users to modify the underlying program in more than one representation. For example, the ALVIS Live! system (Figure 2.4) allows novice programmers to visualise program execution by first visualising the program using graphical representations of data structures, and then allowing them to directly manipulate the program by modifying output and adding new commands. A study with undergraduate students showed the tool was successful in facilitating participants' development of algorithmic solutions (Hundhausen & Brown, 2007).



```
set v1 to 45
create array a1 with 8 cells
populate a1 with random ints between
set i1 to index 0 of a1
while i1 < cells of a1

  add 1 to i1
endwhile
```

Figure 2.4: Novice Environments with Multiple Representations - ALVIS Live!



```
FORWARD 275
RIGHT 40
square 136
LEFT 6
square 168
LEFT 165
FORWARD 80
LEFT 15
REPEAT 18 [
 square 61
 LEFT 18
]
FORWARD 80
LEFT 65
FORWARD 85
LEFT 30
FORWARD 145
TO wheel s2
 REPEAT 36 [
  square 52
  RIGHT 10
 ]
```

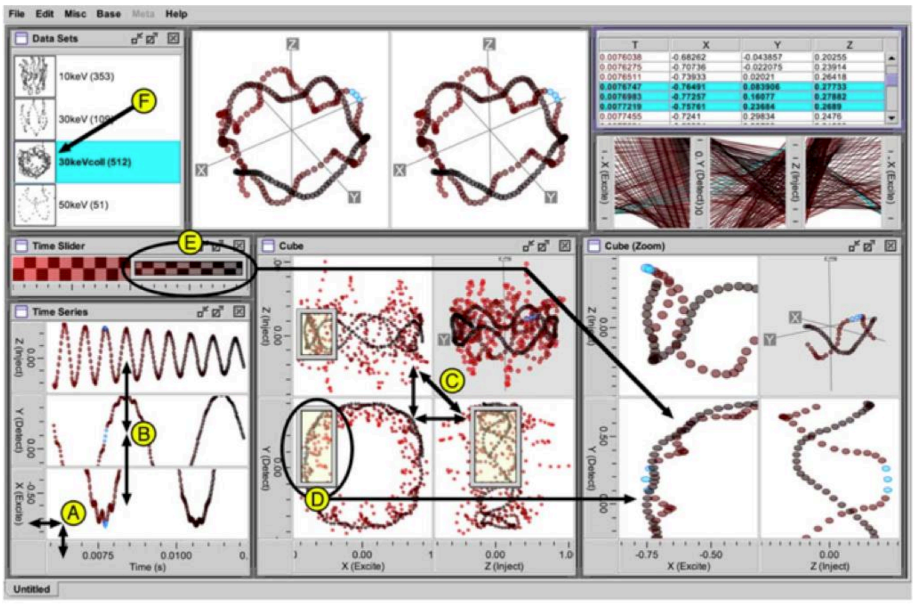
```
RIGHT 36
```

Figure 2.5: Novice Environments with Multiple Representations - Leogo

The Leogo system (Figure 2.5) is another example of a system using multiple representations with an interactive execution environment. Leogo extended a traditional text-based LOGO system to allow the user to modify or create new commands by directly manipulating the turtle graphic at a given point in execution time. It also provided an iconic representation, which allowed the user to specify commands using sliding parameter adjusters. All three representations appear on screen at once and propagate change to one another when edited. A usability study by Cockburn and Bryant (Cockburn & Bryant, 1997) showed that participants enjoyed using the system, but typically used the same representation for a complete task, and avoided using the text representation. This result is confirmed in Chapter 6 of this thesis, and explored in Chapter 8.

Professional use of MERs

MERs have also been used in professional software development systems, where the development environment generates an abstract model of the programming code and displays "views" of the code. For example, the environment might display a view showing a list of methods found in a class. Fowler (Fowler, 2008) refers to the process of editing a view as *Projectional Editing*, where making changes to all views requires one to "project" changes to them via the abstract representation. MERs have also been used to develop Computer Aided Design (CAD) scripting environments that allow architects to view multiple representations of their model (Maleki, 2013).

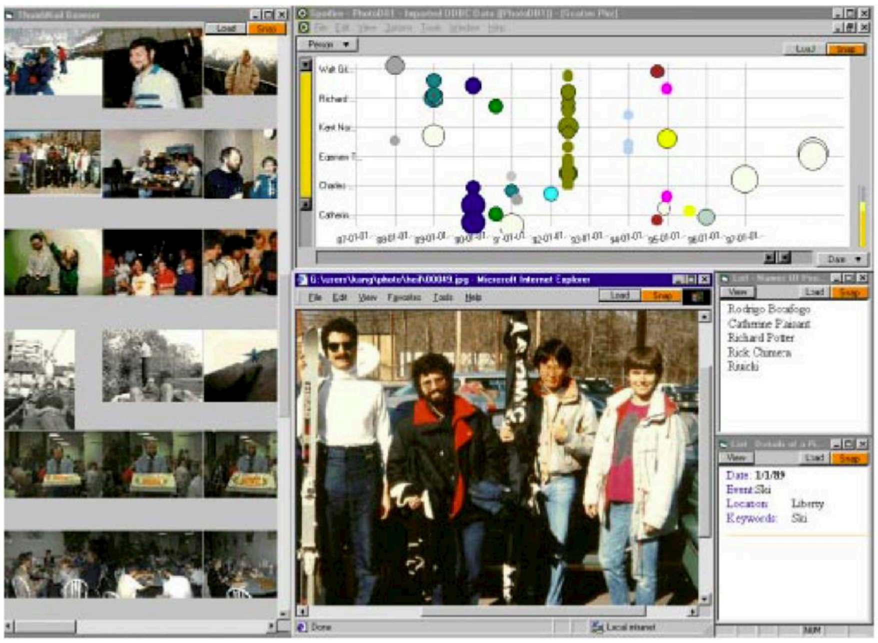


T	X	Y	Z
0.0076038	-0.68262	-0.043857	0.20255
0.0076275	-0.70736	-0.022075	0.23914
0.0076511	-0.73933	0.02021	0.26418
0.0076747	-0.76491	0.083906	0.27733
0.0076983	-0.77257	0.16077	0.27882
0.0077219	-0.75761	0.23684	0.2689
0.0077455	-0.7241	0.29834	0.2476

Figure 2.6: MERs in Visualisation Systems - The Improvise system

Visualisation Coordination

External Representations are often used to visualise data sets or cartographic information. Multiple coordinated external representations have been used in visualisation systems such as Snap (North & Shneiderman, 2000) and Improvise (Weaver, 2004), where users are able to construct MERs by mixing and matching representations to explore the data set. The primary goal of these systems is to facilitate data exploration for expert users. In fact, a study comparing performance of programmers and data scientists using Snap (see Figure 2.6) found that data scientists performed best using dynamically linked representations. It is likely that the expertise of these users allowed them to choose more task-appropriate representations to explore. The Improvise system (see Figure 2.7) allows one to construct MERs using a visual language, which can be modified during data exploration to provide further insight. Both Improvise and Snap coordinate representations live, reducing latency between interaction and feedback as much as possible.



Rodrigo Bocafogo
Catherine Plaisant
Richard Potter
Rick Chimera
Riachi

Date: 1/1/89
Event: Ski
Location: Liberty
Keywords: Ski

Figure 2.7: MERs in Visualisation Systems - The Snap system

2.5 Chapter Summary

This chapter presented a review of theories of cognitive development that have influenced the design of educational programming environments and methods for their use in school environments. Cognitive limitations, such as difficulty transferring fragile knowledge to new contexts are also discussed, followed by a review of how teaching for transfer affects the development of educational programming environments.

A review of existing educational programming environments is also presented, paying particular attention to the different kinds of representations used in each environment, their usability trade-offs, and how they might be used in combination to encourage transfer and reduce the barriers to exploring new representations. Finally, a review of existing Multiple External Representation (MER) environments, spanning several domains, including novice programming, professional software engineering, mathematics and visualisation is presented. Although a small number of MERs have been created for novice programmers, exploration of their use in modern educational programming environments appears to be an underexplored area of research, and will be further explored in the next chapter.

Chapter 3 Developing Notational Expertise in Programming

The previous chapter reviewed theories of cognitive development that influence the design of educational programming tools and their effectiveness in school environments, paying particular attention to the kind of representation used in each tool. The chapter then examined prominent theories of representation and multiple representations, followed by a survey of existing Multiple External Representations (MER) systems in programming education and elsewhere.

This chapter extends theories of representation presented in the previous chapter by reviewing the use of representation in mathematics education, and describing relevant studies that have investigated students' representation translation and transition skill. Parallels are then drawn between campaigning movements in computing and mathematics; in particular, I will argue that the New Math of the 1960s and Computational Thinking (CT) are both centrally concerned with the acquisition of abstract mental operations at the expense of enabling students to develop expertise with abstract representations, or notations, in their respective subjects. Following this discussion, each movement will be used to describe an analogical framework of representation that draws attention to the kinds of representations used in education and their levels of abstraction. Further, I will describe how the framework can be used to i) categorise the programming systems, including those presented in the previous chapter, and ii) identify representation transition strategies. Finally, I will discuss how these strategies can be used to guide design heuristics to improve new programming tools, and thereby help students develop *Notational Expertise (NE),* allowing them to connect conceptual knowledge of programming via the use of MERs.

3.1 Representation use in Mathematics

Mathematics education has been greatly influenced by the theory of constructivism and Bruner's spiral curriculum model (Bruner, 1960), described in the previous chapter. This model of curriculum advocates the introduction of concepts via the sequential use of Bruner's three modes of representation: *enactive, iconic and symbolic.* In mathematics education, students often enactively manipulate objects such as buttons, fruit or blocks to learn concepts such as numbers, addition, subtraction and multiplication. More sophisticated concepts such as division *can* be taught using enactive manipulation of objects, but may become difficult and inefficient to calculate with large numbers, necessitating a transition to symbolic manipulation. Students are encouraged to develop competence in symbol manipulation during primary and secondary school, where they learn about letters, words and grammatical structure in English lessons, and numbers, algebra and calculus in Mathematics lessons.

Students can find symbolic representations difficult to use for several reasons. Before discussing these reasons, it is worth asking what the exact definition of a symbol is, and what properties of symbolic manipulation make them difficult to learn? According to Bates, symbols are interpersonal conventions used intentionally to convey meaning (Bates, 1979). They are primarily used to mediate communication by providing a common reference for conceptual knowledge held in the brain. Unlike icons or indexes, they have no resemblance or relationship to the object they are representing.

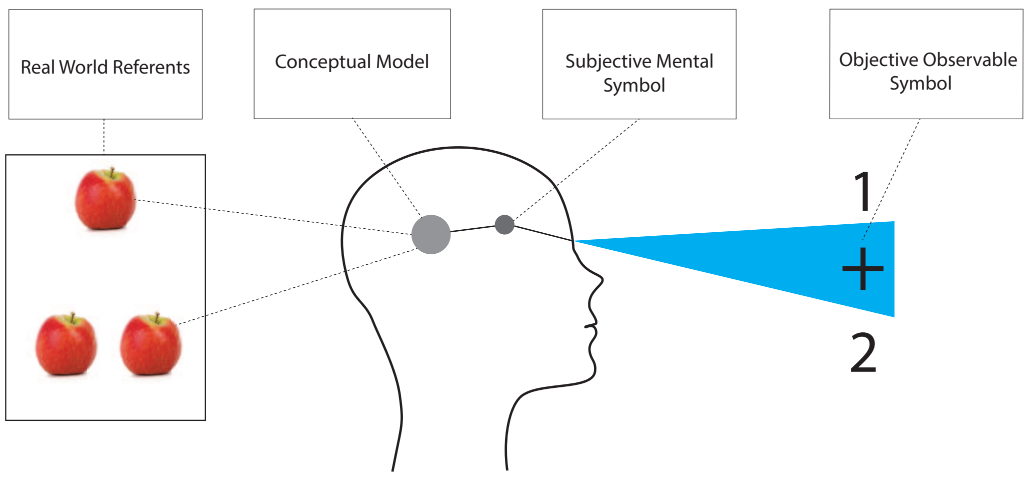


Figure 3.1: Symbols and Referents in the Mental and Physical World

As symbols bear no resemblance to the object or concept they represent, their meaning cannot be known without teaching, or a key. Symbols can also represent ambiguity, or multiple objects at once, resulting in a high level of abstraction. For example, in the equation $x^2 - 4 = 0$, the symbol x can have the value 2 or -2.

Learning new symbols requires a significant effort from the student; in order to use a new symbol, a student must create or possess a mental model of the concept that the symbol relates to, and then store a mapping of the symbol to that conceptual model. Incorrect mapping and poorly constructed conceptual models can cause difficulty for students. The design of the representation system can also cause difficulties; for example, representations with symbolic synonyms, where more than one symbol relates to the same concept (e.g. Figure 3.2), may confuse students. Symbolic homonyms, where a symbol is used to refer to more than one concept, have also been found to cause difficulties in mathematics (Knuth, Stephens, McNeil, & Alibali, 2006), and programming (McIver & Conway, 1996). Figure 3.1 illustrates the connection between the conceptual model of addition, examples of real world referents for addition, the subjective mental symbol for addition and the physical '+' symbol.

$$5 \div 10 \quad 5 : 10$$
$$5 / 10 \quad 10\overline{)5}$$

Figure 3.2: Examples of Symbolic Synonyms in Mathematics

After students learn to identify and interpret symbols, they are taught to manipulate them to solve problems. This manipulation requires well-developed mental models, and the reconciliation of mental operations with appropriate symbols. It can therefore be problematic, particularly if learning is initiated too early. Anghileri uses the example of division in mathematics to illustrate this (Anghileri & Beishuizen, 1998), stating that existing mental operations such as addition and multiplication must already be developed, and that students must be able to relate those operations to the division symbol to complete the division operation accurately. Anghileri also notes that in The Netherlands, the use of symbolic manipulation in division is intentionally delayed to allow the learner to develop a solid understanding of mental strategies for division. McIver and Conway identify similar symbol manipulation issues in programming and propose design principles to reduce such problems, including making a clear separation between syntax and semantics, and making syntax more readable and consistent (McIver & Conway, 1996).

Mental Strategies for Symbolic Manipulation

Programmers and mathematicians must both ultimately develop symbolic manipulation strategies to create working programs and find mathematical solutions. Lampert distinguishes between two strategies for symbolic manipulations in multiplication: *procedural*, where students use concrete, computational strategies that can be rote learned and applied in steps; and *principled*, where students have developed the conceptual knowledge to allow them to invent appropriate mathematical procedures (Lampert, 1986). In agreement with Anghileri, Lampert states that schools teach procedural methods too early, and promote "mindless" manipulation of symbols, severing the connection between symbol manipulation and mathematical concepts. Nunes studied Brazilian street sellers to show that when this connection is severed, participants "mindlessly" rely on procedural methods, decreasing their performance when compared with the equivalent verbal calculations (Nunes, Schliemann, & Carraher, 1993).

Representation Translation and Transition

Some researchers argue that representation is central to mathematics and that representational knowledge mediates complex problem solving skill (Heritage & Niemi, 2011). To measure conceptual knowledge and problem solving skills, new assessment methods have been proposed based on students' ability to use more than one representation to solve a problem. In light of a study by Niemi (Niemi, 1996) showing that higher representational knowledge resulted in increased understanding and problem solving skill, Heritage presents a cyclical framework that allows teachers to formatively assess and iteratively adapt to student knowledge by analysing the representations they create (Heritage & Niemi, 2011).

Other researchers have investigated the importance of translation between representations in the classroom. In semi-structured interviews with mathematics teachers for the purpose of identifying teaching practices for translation between representations, Bossé et al (Bossé, Adu-Gyamfi, & Cheetham, 2011) showed that teachers acknowledged that students found some representation translations, such as graphical to algebraic, easier than others, but that targeted teaching strategies could help to improve difficult transitions. Gagatsis found that translation ability was an important factor in mathematical problem solving ability during an examination of student representation translation ability by asking participants to translate functions from verbal form to graphical and algebraic, and then from graphical form to verbal and algebraic. Gagatsis concluding that true mathematics is the ability to recognise an idea embedded within a representation, manipulate the idea within that representation, and translate the idea from one representation to another (Gagatsis & Shiakalli, 2004).

Nathan et al. extended this work by investigating students' ability to solve problems and translate between tabular, graphical, verbal and symbolic representations (Nathan, Stephens, Masarik, Alibali, & Koedinger, 2002). The study found that students using a single representation experienced more success, but that success was strongly influenced by the representational 'formats' involved. Nathan concludes that students are likely to attain fluency with instance-based representations with limited abstraction before making the transition to highly abstract, 'holistic' representations.

3.2 Significant Movements and Representation

This section presents two campaigns in mathematics and computing that I will argue have been significantly concerned with representation.

3.2.1 The New Math

In the 1960s, a movement called "New Math" began with the aim of increasing rigour and understanding in mathematics. Introduced by a group of French mathematicians under the pseudonym Nicholas Bourbaki, New Math encouraged the teaching of rigorous abstract mathematics concepts, such as set theory, from a young age (Adler & Adler, 1959). In an effort to boost mathematical understanding, students beginning mathematics education would be encouraged to use highly abstract set terminology (e.g. the union of a set with cardinality of 2, and the set with cardinality of 1 results in a set of cardinality 3) instead of first using concrete representations with low-abstraction (e.g. two apples and one apple make three apples) (Kline, 1973).

Critics of the New Math argued that in order to learn mathematical concepts, students had to be shown the connection to the real world. Kline illustrated this point by stating that the failure to link mathematics to real world meaning was "analogous to teaching students how to read musical notation without allowing them to play music", and went on to state that New Math would result in mathematics becoming pointless and unattractive (Kline, 1973). Feynman added that textbooks based on the New Math limited student flexibility of choice in the method used to solve a problem, denying effective non-rigorous approaches such as trial and error (Feynman, 1965).

Although the New Math supported conventional external representations used in traditional mathematics such as graphical, tabular, tangible and verbal, it placed emphasis on symbolic representation by promoting the use of rigorous deductive development using axioms and proofs. New Math also encouraged highly abstract use of other representations in order to allow students to develop sophisticated internal representations and cognitive operations. As discussed in the previous chapter, such representations have been predicted to be more difficult for students to compute due to the number of cases that need to be considered (Stenning & Oberlander, 1995) . Furthermore, students may find it more difficult to build new internal representations if they are not sufficiently associated with familiar concrete examples.

3.2.2 Computational Thinking

The second of the two campaigns is Computational Thinking (CT), which was introduced by Wing in 2006 (Wing, 2006) with the intention of popularising computer science and enabling non-computer scientists to develop computational solutions to problems within their domain. Wing describes CT as the development of the "*thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out*" (Wing, 2014). Although the precise definition of CT has been widely debated (The National Academy of Sciences, 2010), in her original paper, Wing is explicit in the view that CT is not about the use of notations, but about the development of abstract mental tools such as problem decomposition, pattern generalisation and abstraction.

To develop mental tools for CT, including those proposed by organisations concerned with education (Google, 2013; ISTE, 2013), researchers invited to a workshop on the pedagogical aspects of CT (The National Academy of Sciences, 2011) suggested several strategies: Tinker advocated the use of modelling and simulation environments such as NetLogo and Agentsheets for developing CT with K-12 students in order to minimise time mastering programming An alternative view, provided by Resnick and Kolodner in the same workshop, stated that CT requires that the thinker should be able to create, build and invent representations using computation, which requires fluency with computational media. Finally, with regard to representation, Collins went further, and argued that CT should emphasise the development of representational competence.

In her original paper, Wing encourages professors to teach courses that demonstrate "ways to think like a computer scientist" to allow students from non-computing disciplines like science and engineering to apply CT tools to their work. However, computer scientists in training inevitably do some kind of programming activity during an undergraduate degree, allowing them to develop and refine the sophisticated mental models required to support CT tools. I therefore contend that the view held by Resnick and Kolodner is correct, and that students would be able to build better mental tools for CT by achieving fluency with notations. I also suggest that the level of abstraction in such representations may affect the ease with which students can learn using programming representations.

3.3 Modes of Representational Abstraction (MoRA)

The core theoretical motivation for this thesis is based on this section, which draws an explicit comparison between the ideas of New Math and Computational Thinking to describe the

Modes of Representational Abstraction (MoRA) framework, which can be used to categorise existing representations used in programming and mathematics education. This framework will be used to identify suitable representation transition strategies that can inform the design of new education programming tools.

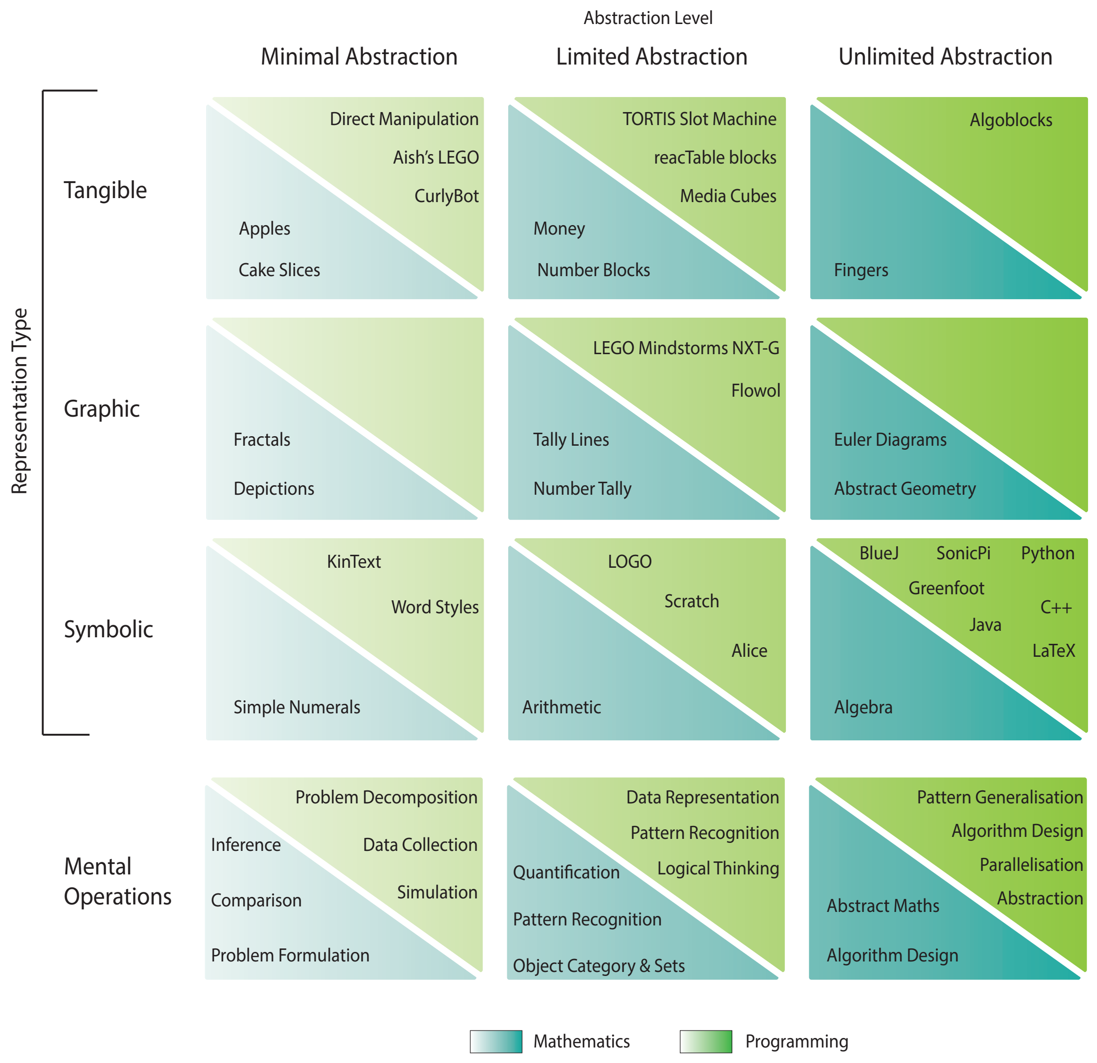


Figure 3.3: MoRA: An Analogical Framework of Representation Use in Education

The vertical axis distinguishes between representation type: tangible, graphic and symbolic, inspired by Bruner's modes of representation model. The horizontal axis distinguishes between the level of abstraction supported by the representation using Stenning and Oberlander's classification of abstraction support in representation systems. The final row depicts mental operations related to each column, including those described in CT.

The MoRA framework, shown in Figure 3.3, presents 9 different categories of representation that are each split into two, with one half describing programming representations and the other describing mathematics representations. The framework simplifies the direct comparison of representation types in mathematics and programming, and provides a context for which New Math and Computational Thinking can be used to illustrate existing representation transition strategies and why they can be problematic. The framework categorises representations using two dimensions: the vertical axis, which is used to distinguish between types of representation, such as tangible, graphical and symbolic, and the horizontal axis, which is used to indicate each representation's facility to support different levels of abstraction.

3.3.1 Types of Representation

The vertical axis relates to the type of representation used in the educational system. This axis contains three external representations common to both mathematics and programming education: tangible, graphic, and symbolic, which are tightly coupled to Bruner's three modes of representation (Bruner, 1960).

Comparable to "manipulables" in mathematics (Pimm, 2002), **tangible** representations are physical objects with properties, such as position, configuration and orientation, that can be manipulated. Tangible Programming Languages have existed since the introduction of AlgoBlocks two decades ago (Suzuki & Kato, 1995), with Tangible User Interfaces being defined soon after (Ishii & Ullmer, 1997).

Graphical representations are two dimensional objects that consist of a *graphical vocabulary*, consisting of individual marks or components, and *graphical organisation*, which describes the way each component is related to one another (Blackwell & Engelhardt, 1998). Visual programming languages (VPLs) are often described by their graphical properties. However, a method for categorically distinguishing between graphic and sentential representations is yet to be agreed on (Shimojima, 1999). Some VPLs intentionally use a similar visual structure to text languages, whereas others behave more like diagrams. This framework will categorise VPLs according to how symbolic or graphical they are, rather than follow a fixed definition.

Symbolic representations are interpersonal conventions used intentionally to convey meaning (Bates, 1979). Refer to previous sections for a detailed discussion.

The MoRA framework provides an additional row called **Mental Operations,** which describes operations used to manipulate internal representations. This term is used by Piaget, who states that pre-adolescent children will develop mental operations such as sorting, classification, reversibility, and conservation during the concrete-operational stage of development, and abstract thinking, hypothesising and deduction during adolescence at the formal-operational stage (Piaget & Inhelder, 1969). Zhang also makes reference to similar skills, but calls them "cognitive operations" (Zhang, 1997).

3.3.2 Levels of Abstraction

The horizontal axis in the framework relates to the level of abstraction a representation can support. These levels are adapted from Stenning and Oberlander's three levels of abstraction in representational systems, namely minimal abstraction (MARS), limited abstraction (LARS) and unlimited abstraction (UARS) (Stenning & Oberlander, 1995).

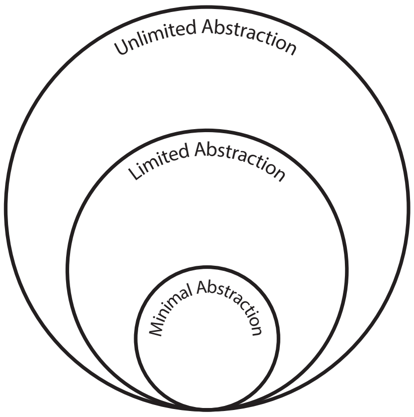


Figure 3.4: Levels of Abstract in Representation

MARS is the simplest type of representational system; representations have minimal abstraction if there is exactly one model that corresponds to a particular representation in the system. For instance, in the mastermind code-breaking game, a row of letters can be used to represent the colour of pawns, such as [R R B Y Y]. These letters represent a single state in the game.

LARS are minimally abstract representations that permit some object to take more than one value in a certain dimension. For instance, in the mastermind game, the row of letters could contain a symbol like '-' to represent an undetermined colour [R R – Y Y]. This representation describes any world where a pawn of any colour is situated between two red pawns and two yellow pawns.

UARS are systems that express an unlimited number of worlds via dependencies inside the representation using equations, or key assertions to outside representations.

3.3.3 The New Math

The primary emphasis of the New Math was to encourage abstraction and rigour when describing, and carrying out operations on, mathematical objects (Adler & Adler, 1959). This emphasis spanned tangible, graphic and symbolic representations and can therefore be mapped to the UARS column of the MoRA framework (Figure 3.5).

Although New Math illustrated mathematical ideas using each type of representation described in the MoRA framework, it encouraged learners to manipulate the representations symbolically. For example, in the first chapter of his book on the New Mathematics, Adler uses the example of mapping days of the week to fingers on a hand to demonstrate matching operations between two sets, followed by the introduction of new symbolic notation to describe this mapping (Adler & Adler, 1959). In the same book, Adler later used graphic representations to describe the translation of triangles on a set of axes, but quickly went on to describe the movement using symbolic notation:

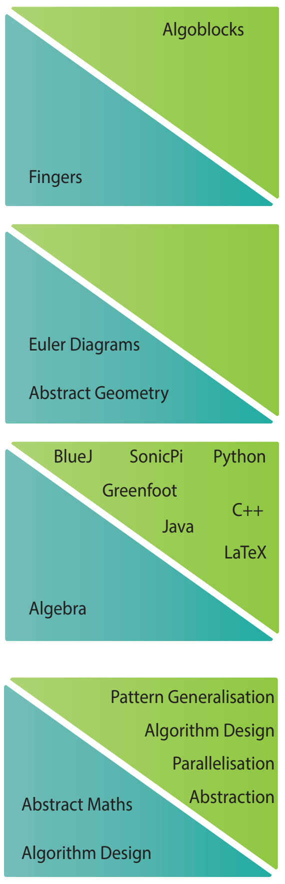


Figure 3.5: The New Math in the MoRA Framework

Top: Tangible UARS, **Second from Top**: Graphic UARS, **Second from bottom**: Symbolic UARS, **Bottom**: Mental Operations.

"We now have a system consisting of six elements, I, P, Q, R, S and T. We define a binary operation * for the system as follows…". The large number of cases described by UARS may act as a barrier to learners, as an increase in cases is predicted to increase computation (Stenning & Oberlander, 1995). Critics have argued that the abstract nature of New Math caused it to focus on "hopelessly artificial" problems and thereby reduce emphasis on developing competancy with representations more likely to be used in day to day life (Kline, 1973).

What type of mental tool should be developed?

The New Math encourages learners to develop complex mental operations without allowing them to build competence with concrete representations. The view that this is likely to cause difficulty for students learning mathematics is shared by Anghileri and Lampert, who as described earlier, believe that introducing symbolic manipulation without developing robust mental operations severs the connection between symbol manipulation and mathematical concept. A more effective strategy might be to encourage learners to begin developing core mental operations using tangible representations, and increase the level of abstraction over time.

3.3.4 Computational Thinking

As explained previously, CT advocates the development of mental strategies that allow people to form computational solutions to problems. These strategies can be mapped to the "mental operations" row, which presents tools for manipulating representations in the MoRA framework.

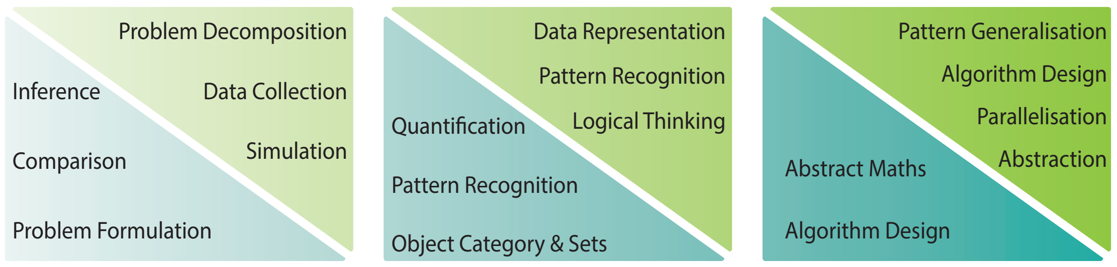


Figure 3.6: Computational Thinking in the Representation Transition Framework

Do they exist at each level of abstraction?

The framework places a collection of CT mental operations at each level of abstraction (MARS, LARS and UARS). The position of these mental operations loosely denotes the abstraction level of representation each tool can be used to manipulate. For example, data collection is typically the process of collecting minimally abstract representations (MARS), pattern recognition requires identification of parts of a representation that can be replaced by a single representation (LARS), and algorithm design is the process of generating a representation that can apply to an unlimited number of worlds (UARS).

Which strategies should be considered?

Wing states below that appropriate representations must be chosen to develop CT strategies. However, researchers do not agree on the types of representations that should be used, the number of representations, the level at which they should be introduced, and how long they should be used.

"C.T. is choosing an appropriate representation or modelling the relevant aspects of a problem to make it tractable" (Wing, 2008)

Stenning and Oberlander argue that representations with high abstraction require a greater amount of computation due to the increased number of cases that have to be considered (Stenning & Oberlander, 1995). In order to reduce the number of cases that have to be considered, and therefore the computation effort required, learners may benefit from first developing mental operations to manipulate representations that support minimal abstraction (MARS).

Representations and cognitive development

Discussants in a workshop on pedagogical aspects of CT organised by the National Academy of Sciences stated that it remains a challenge to identify the age or grade level at which students can handle abstraction, but that interactive visualisations or simulations lay at the heart of Computational Thinking (The National Academy of Sciences, 2011).

If we agree that to reduce computational barriers, learners should begin by developing mental operations to support MARS, which type of MARS should learners start with, and in what sequence should they progress? Piaget stated that in the Concrete-Operational stage, children are only able to apply developed mental operations to concrete representations, such as oranges or trees (Piaget & Inhelder, 1969). Bruner agrees that children first develop processing systems to manipulate enactive representations, and that they are a useful starting point for learners of any age (Bruner, 1960). In light of these views, it would be preferable for learners to develop these mental operations first by using tangible representations, building on them by manipulating graphic and symbolic representation over time.

3.3.5 Education Programming Systems

Conventional programming representations are typically text-based, and are used to create abstract, general-purpose programs that can deal with a range of inputs. They are therefore placed in the most difficult category for learners to access: Symbolic UARS. Unfortunately, the majority of text-based educational programming systems described in the previous chapter do not fare much better; BlueJ, SmallBASIC, and SonicPi, although highly supported, are all categorised as Symbolic UARS due to their use of general purpose programming languages.

Some educational systems using VPLs are also categorised as symbolic, due to their use of text. However, these VPLs are categorised as Symbolic LARS, rather than UARS, as the complexity and structure of text that can be specified is limited. The languages are also typically limited in the types of inputs they can process. For instance, Scratch requires users to declare a fixed number of variables before the program runs, and does not allow users to programmatically access the file system.

Graphical systems such as LEGO Mindstorms NXT-G and Flowol also limit abstraction, but may be more accessible to learners than symbolic representations if perceptual affordances, such as relative position, grouping of objects, size, colour and orientation, are used appropriately (Engelhardt, 2002)[1].

Tangible educational systems such as Perlman's TORTIS Slot Machine, are categorised in the MoRA framework as Tangible LARS as they provide learners with the opportunity to physically manipulate parts of the representation, but are powerful enough to specify many different programs.

Programming systems with MERs can span multiple cells in the framework; when introducing their cognitive theory of graphical representations, Stenning and Oberlander defined LARS using the example of two slightly different MARS placed side-by-side. The MERs systems described in the previous chapter, such as ALVIS Live! and Leogo, contain both symbolic UARS and graphical LARS.

3.4 Transition Strategies to support Notational Expertise

This section will use examples from the MoRA framework to identify representation transition strategies that would support novice programmers in building robust mental operations that can be used to manipulate abstract representations, or notations, such as those found in conventional programming environments.

The placement of New Math in the MoRA framework, and work presented earlier in the chapter, indicate that it would be a mistake to encourage students to use abstract representations too early, particularly if students have not reached the Concrete Operational stage of cognitive development, which Piaget believed was between the ages of 7 and 11.

The placement of Computational Thinking in the MoRA framework indicated that different mental operations might be developed for manipulating different types of representation, and

[1] It is also possible for computational load to increase compared with text-based languages (Sweller, Chandler, Tierney, & Cooper, 1990)

that acquiring the CT abilities described in the literature would require the use of different types of representation over a range of abstraction levels.

The placement of educational MER systems in the MoRA framework illustrates that systems can span more than a single cell – indeed, this is a promising approach that may help students to make the transition between representations. However, research in mathematics discussed at the start of this chapter also shows that care must be taken to introduce each representation at an appropriate time, and for an appropriate task. Further, multiple representations can be used to assess students' conceptual knowledge, which could provide the basis for measuring effectiveness of a MER system.

3.4.1 Transition Strategies and Notational Expertise

Several possible strategies exist for making the transition to symbolic UARS. Figure 3.7 shows one such transition strategy, which would begin with learners using low-abstraction tangible representations such as CurlyBot – a robot that can be manipulated to create executable movement using Programming by Demonstration (PbD), demonstrating program execution, and Direct Manipulation (Frei, Su, Mikhak, & Ishii, 2000). Learners could then use this knowledge to move to tools with increased levels of abstraction levels, such as the TORTIS Slot Machine (Perlman, 1976), in which they could specify programs by placing physical cards into slots in sequence. An understanding of execution and program flow could then facilitate the transition to limited-abstraction graphical systems such as Flowol, which allow programs to be specified using flow diagrams. An understanding of graphical objects and execution flow could then be used to make the transition to limited-abstraction symbolic systems, such as Scratch and LOGO. Finally, learners could be encouraged to move to highly supported educational environments with unlimited-abstraction, such as Greenfoot.

However, there are considerable costs associated with using many different systems; students must develop new or existing mental operations in order to deal with each representation, learn a set of new semantics, and learn to use features of the system not associated with its programming representation.

It is known from previous chapters that the use of more than one representation can help learners in several ways. But what kind of skill, if any, can a learner develop when using those representations? In both Mathematics and Programming, expertise is demonstrated through the understanding of notations – a term used in this thesis to describe abstract representations. One might say that when learners have acquired sufficient knowledge to understand a particular concept, and have developed suitable mental operations to manipulate notations of that concept, they develop Notational Expertise *(NE)*. The properties of NE listed below extend observations made by Gagatsis (Gagatsis & Shiakalli, 2004), Ainsworth (Ainsworth, 2006) and Lesh (Lesh, 1999). To develop NE, one must:

1. Master the use of more than a single notation
2. Recognise ideas embedded within a notation
3. Identify correspondences between notations
4. Translate from one notation to another
5. Understand the suitability of a notation for a particular task
6. Be able to construct notations

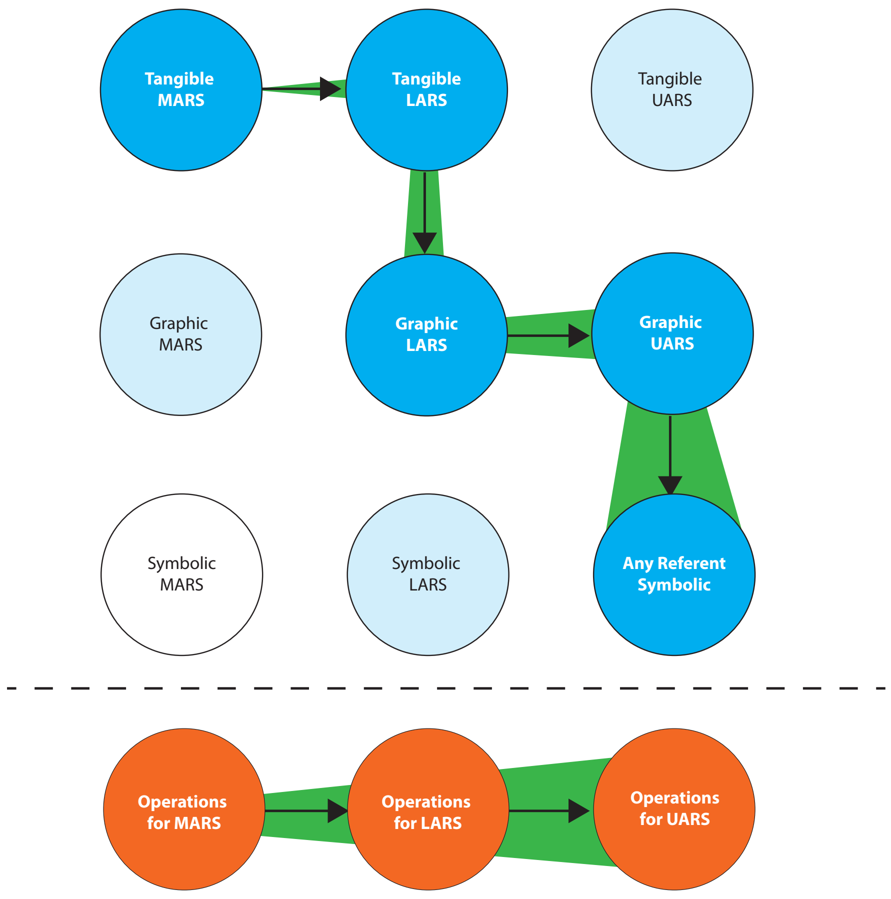


Figure 3.7: Proposed Educational Strategy for Representational Transition

3.4.2 Acquiring Notational Expertise with New Programming Tools

Previous work has shown that MERs may provide many benefits to learners. They can emphasise different properties of the represented world, support switching to reduce the computation effort required, support computational offloading to reduce cognitive load, and if required constrain the number of inferences that can be made from other representations.

However, these benefits are highly dependent on the choice of representations and their suitability to the required task. The cost of transition may also change depending on the direction in the MoRA framework – either between levels of abstraction or between types of representation; if the student has no experience with graphical representations, a transition from tangible to graphical may be more difficult than an increase in abstraction in already familiar tangible representations.

I propose that new kinds of MER educational programming environments should be developed to investigate the benefits of transition strategies such as those described above. Such MER systems may be able to facilitate students in making transitions between representations by reducing the cost associated with each transition. In a single integrated MER system, students would only be required to use one system, where there is a single, fixed set of semantics. It may be possible to design representations such that each could support different functions or levels of abstraction. The order of representations could also be fixed or modified depending on the needs of the student. Students may benefit from being able to return to previous representations if they experience difficulty with difficult or highly abstract representations, or want to validate their existing mental operations.

3.5 Chapter Summary

In summary, I have presented two educational movements used to illustrate representation transition strategies within the Modes of Representational Abstraction (MoRA) Framework. The MoRA framework classifies mathematics and programming representations according to their type, and level of abstraction. Lessons from each educational movement assisted the identification of improved representation transition strategies, which would allow learners to move from familiar, tangible representations with low abstraction, to symbolic representations with high abstraction. Following the identification of this strategy, I defined *Notational Expertise*, which provides a set of criteria to measure students' conceptual knowledge. Finally, I proposed the use of MER programming environments that could help to reduce the cost of representation transition, and provide high levels of support for learners.

Chapter 4 Teaching Interviews

4.1 Introduction

The analogical framework presented in the previous chapter provides a mechanism for categorising computing education tools based on abstraction level and mode of representation. This chapter investigates the use of such computing education tools to support student understanding of computing concepts and practice within a classroom environment.

I carried out ten semi-structured interviews and subsequently conducted thematic analysis in order to investigate current teaching practices. Participants consisted of eight professional teachers and two professional software developers who had experience with teaching in after-school clubs. In addition to the interview, I was also able to elicit expert feedback on an early prototype of a novice software development environment I had developed. Interview participants had varying levels of computing and teaching experience, in addition to familiarity with diverse school environments and levels of funding (see Table 4.1).

The study had four aims: (1) to investigate current teaching practice and tools, (2) to investigate the main barriers to using new tools, (3) to investigate the main barriers to student progression, and (4) to elicit early feedback on an educational software development environment.

4.2 Research Methods

This study used a two-part interview; the first part was semi-structured, posing questions relating to current educational software tools, teaching practice, student feedback and use of assessment (see Appendix A). The second part of the interview elicited feedback on an early prototype of a new educational development environment. Semi-structured interviews generate a rich set of data, and allow participants to express views in their own terms. Guide interview questions were generated from design considerations and observations, which I recorded during attendance of a Continuous Professional Development (CPD) workshop.

4.2.1 Participants

The study included two kinds of participant. The first were ICT and Computing teachers recruited via direct contact with schools, Computing at Schools (CAS) meetings and CPD workshops. Four participants taught at non-selective state secondary schools, three participants taught at selective private schools, and one participant taught at a selective state secondary school. The second were professional software developers who had experienced teaching computing in after school clubs as part of CodeClub, a volunteer-led network of coding clubs for 9-11 year olds.

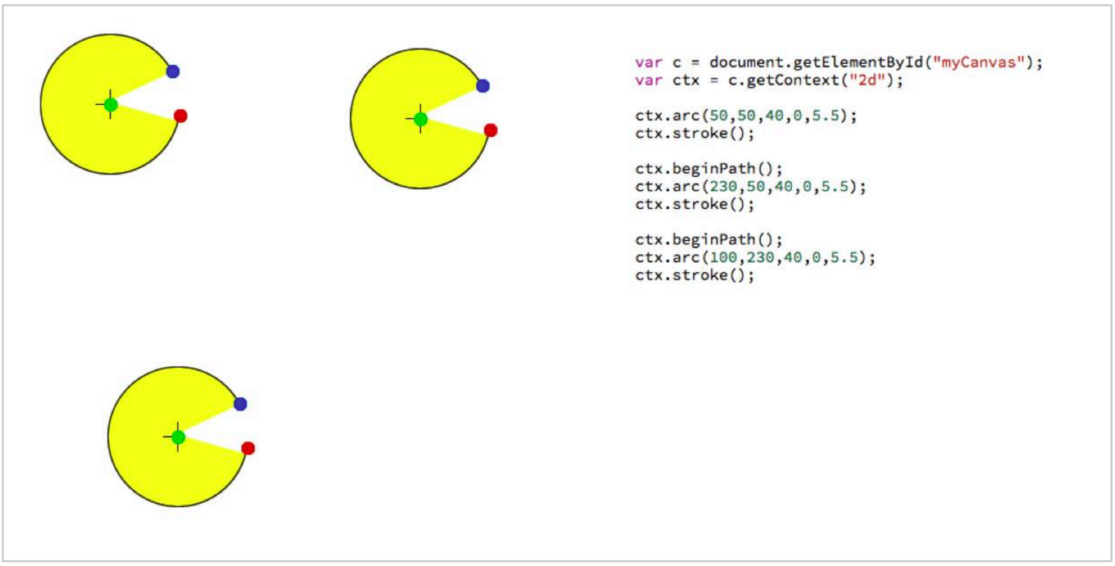
Table 4.1: Summary of Teacher Background and Experience

Pseudonym	**Background**	**Role**	**Teaching Experience**	**School Type**
Heather	Bsc Geography	Head of IT	> 20 years	Independent
Richard	BA German	IT Teacher	10 years	Independent
Mary	Bsc CS, Industry	Head of IT	15 years	Independent
Andrea	Bsc CS	IT Teacher	5 years	State
Brian	Bsc CS, Bsc Mathematics	IT Teacher	20 years	State
Mark	Bsc CS	Head of IT	10 years	State
Keith	Bsc CS	IT Teacher	5 years	State
Stephen	Bsc CS	IT Teacher	5 years	State
Margaret	Bsc CS	Software Engineer	1 year (Code Club)	State
Ben	Bsc Physics	Software Engineer	1 year (Code Club)	State

4.2.2 Procedure

Teaching participants were interviewed in their place of work where possible. Each interview lasted approximately an hour and a half, with one hour focusing on semi-structured interview questions and half an hour focusing on a prototype demonstration and feedback questions. Software developer interviews used the same procedure, but took place in participants' homes.

The demonstration showed an early software prototype, which contained an example of bi-directional translation and live programming execution. The example showed the JavaScript code to draw a portion of a circle. The output was displayed in a Direct Manipulation panel, in which properties of the circle, such as position and radius could be modified directly (Figure 4.1).



```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");

ctx.arc(50,50,40,0,5.5);
ctx.stroke();

ctx.beginPath();
ctx.arc(230,50,40,0,5.5);
ctx.stroke();

ctx.beginPath();
ctx.arc(100,230,40,0,5.5);
ctx.stroke();
```

Figure 4.1: Prototype Educational Software Environment

4.2.3 Data Collection and Analysis

In order to identify common features and themes in participant responses, I employed a concept-driven coding methodology (Gibbs, 2007), which extends the ideas of framework analysis and template analysis (King, Cassell, & Symon, 2004). This style of coding requires that researchers generate a set of codes based on previous research, conceptual frameworks in the literature, and a selection of transcripts. The approach recognises that new, emergent codes should be added to the scheme when required. *A priori* codes were generated from the Cognitive Dimensions of Notations framework (Green & Petre, 1996a), which describes properties of notation used to evaluate suitability for a given task, well established programming activities (Green & Blackwell, 1998), Computational Thinking abilities (Wing, 2006), and CS Education literature describing difficulties of using new software in classrooms (Bauer, 2005; Bingimlas, 2009).

The procedure for this study was approved by the Computer Laboratory Ethics Committee. Each participant gave consent for recordings and subsequent transcripts to be used. Interviews were recorded using a high quality digital voice recorder, and transcribed verbatim. Coding was carried out using qualitative data analysis software.

4.3 Findings

Findings are presented in two stages. In the first stage, an overview of common practice and the barriers teachers experience using educational software is presented. In the second stage, consideration of expert feedback is presented. In both stages findings will be presented using excerpts from participant transcripts.

4.3.1 Theme Identification

After coding the corpus of transcribed interview data, *a priori* and emergent codes were collected and grouped into themes by identifying repetition of similar concepts, comparing similarities and differences of codes, and finally cutting and sorting, as recommended by Ryan and Bernard (G. W. Ryan & Bernard, 2003). Ten transcripts were tagged with 303 instances of 63 codes. Discussion of findings for the first stage of the interview will be presented using the most prominent themes identified during analysis (see Table 4.2). Although the majority of *a priori* codes were used during analysis, some codes were not as popular as expected and therefore were not included in the discussion. For example, Computational Thinking and individual CT abilities were not referred to by any of the teachers directly. There was also less emphasis on assessment than expected.

Table 4.2: Identified Themes

Theme	Source
Linking concrete and abstract representations	Emergent
Representation-specific Difficulties	CDs
Error proneness in text environments	CDs
Reduced visibility and Diffuse notation	CDs
Hard Mental Operations, Diffuseness and Hidden Dependencies	CDs
Non-Representational Difficulties	-
Infrastructure Problems	CSEd Literature
Lack of Supporting Knowledge and New Contexts	CSEd Literature
Documentation and Error Messages	CSEd Literature
Ability Dependent Concepts	CSEd Literature
Teaching Skill	Emergent
Motivation, Liveness and Gender Differences	Emergent
Teaching Approach and Assessment	CSEd Literature

4.3.2 Linking Concrete and Abstract Representations

All teachers except Richard stated that they begin introducing programming concepts to students using oral descriptions and metaphor, or paper-based representations. Examples of algorithm descriptions include running a bath, making a cup of tea, the Sheldon Cooper friendship algorithm[1] and instructions to make a paper airplane. Two teachers, Mark and Stephen, stated they used a shoebox to introduce the idea of variables. Stephen also uses the

[1] A simple algorithm to make friends from the television show "The Big Bang Theory".

metaphor of two islands connecting via a bridge to teach the concept of hyperlinks. Stephen gave the following example of introducing variables:

> *"We talk about variables, and I'll throw a box on the floor and label it with sticker "A" and we'll talk about I've got this piece of data and I want to put it into there. But how do I know what's in there, I have to go and get it and pull it out". (Stephen)*

Brian stated that writing on paper before using programming environments allows students to reflect: *"The paper is just to get them to organise their thoughts. If you jumped straight into Scratch, sometimes they try to do too much – they haven't organised what they want to do".* Heather suggested that paper is a useful tool for planning and Brian stated that*"[paper is useful,] especially at the initial stages we've done things like, we'll be using Scratch and Python and try to implement the same programs and design it on paper first".*

In addition to implementing the same programs with text and visual representations, five of eight teachers stated they used programming representations side-by-side to show correspondences between representations. Those that did use side-by-side representations mainly used Adobe Dreamweaver and encouraged students to use the HTML/Direct Manipulation split-view interface to allow them to see what happened to the code when changes were made in the What-You-See-is-What-You-Get (WYSIWYG) representation. When asked how current tools could be improved, Stephen stated:

> *"I would have loved to see a way for Scratch to be text-based as well. ... To be able to have Scratch in a split-screen mode so they can try to do something on their own in text, and switch back to see the proper code, or if they get stuck, do it in jigsaw piece, and see the code that's generated, so they can see why they're not getting it right, and how to fix it". (Stephen)*

Some teachers, including Mary, mentioned using specific order of tools in order to reduce the problems moving between representations:

> *"In year 9 they do App Inventor and I refer back to Scratch. It was interesting the amount of them that say "oh no I can do that!" [after] I give them a lesson to remind them about Scratch". (Mary)*

Richard stated, "*One of the issues for us, was do we teach python first, do we teach App Inventor first? And the idea is will the student, from doing one, be able to transfer the concepts over to the other*". Another teacher, Brian, stated that he intentionally moved from using LOGO to Python by introducing students to the turtle module in Python and encouraging them to recreate programs originally written with LOGO.

Finally, Mary stated that she regularly used side-by-side comparison to highlight similarities between the Microsoft Windows file explorer and the command prompt:

> *"...we were looking at interfaces. So I just had file manager - windows explorer, and the command window and showing them this here is a folder here, and that's the icon, and double click here. And get them to see that it's exactly the same thing. It was looking at the c drive on both of them, and showing them this is the graphical user interface, is this. That's exactly what they need as far as I'm concerned. To show them that, so they can link it together, rather than having this idea of that's this and that's that, having different models rather than realising they are the same thing."*

In sum, it appears that teachers introduce software tools in a deliberate and specific order, starting with oral or paper-based descriptions, moving to a visual language and then to a text-based language. In some instances, they use side-by-side examples directly to compare two representations, and in other instances refer back to concepts in familiar representations while providing introduction to new representations.

4.3.3 Representation-specific Difficulties

Teachers and software developers used representational difficulties as a filter for their choice of computing education tools to use in the classroom. It became clear that after introducing computing concepts through discussion, or on paper, classroom teaching of computing concepts was supported by use of educational development environments.

Error Proneness in Text Environments

All participants had concerns relating to *Error Proneness* $^{(CD)}$ – a dimension which asks whether the design of the notation induces careless mistakes (Green & Petre, 1996a). One teacher, Mark, stated that "*We did another program with one of the clubs with GreenFoot work, we did a bit of work with methods – they kind of get the idea but really struggled with java syntax*". Another teacher, Brian, said "*The biggest hurdle in Python, is when they make a mistake, things don't happen as they expect, they get a syntax error*". Furthermore, both teachers and software engineers referred specifically to Python's indentation: Keith said "*The whole indentation thing – that's another thing that takes a while*", and Ben said "*And then* *there's the actual typing issues, and the indentation issues, and python's rubbish error generation*".

Mark noted that the error proneness of the representation needed to be explained to students before using the notation:

> *"I just said from the offset we're going to get this wrong, we're going to get this kind of error. If this happens, check your curly braces. I said from the offset - it's going to go wrong. That worked relatively well, there were a few that weren't happy they were being set up to fail." (Mark)*

Two teachers, Richard and Mark, noted that visual languages were used specifically in order to avoid error proneness with younger students "*The idea being at that age that it's pretty and colourful, and there isn't the opportunity to get coding errors, either it runs or it doesn't*", and "*With Scratch, it's obviously very visual. The only things you fill in is [sic] obviously the variables and numbers. You can't really make syntactic errors*".

Reduced Visibility of Diffuse Notation

Teachers and software developers observed that visual languages suffer from reduced *visibility* $^{(CD)}$ of visual notation due to high *diffuseness* $^{(CD)}$, which results in increasing complexity when modifying or debugging programs: Ben said "*Sometimes they know what they're looking for but can't find it – that's a downside of Scratch, if you've got a 1024x648 [screen] they find that a problem*", and Brian said "*When they want to change what's going on on the screen, it gets quite complicated keeping track of everything in the interface. It doesn't seem easy to make a complicated multi-level game*".

Hard Mental Operations, Diffuseness and Hidden Dependencies

Two teachers noted that students find the notation difficult to use when there are *hidden dependencies*$^{(CD)}$ in the notation:

> *"When they start making more complicated things, you start getting into quite high-level concepts like passing messages. Those sorts of things they can cobble something together, but it's quite easy for them to come unstuck doing it." (Brian)*
>
> *"They just see it as "I'm shouting", but they don't see it as targeting…there's no connection, no physical line, they struggle to see where it's going on." (Stephen)*

In addition to hidden dependencies, Brian reports that the number of blocks used by students contributes to the level of difficulty they experience: "*There are so many scripts going on it's hard to keep track of what's doing what.*"

In sum, it appears that representational difficulties are the primary consideration for teachers when introducing new programming environments. Dealing with a high level of errors, which predominantly appear in text-based environments, can cause anxiety, and reduce student confidence. Some teachers attempt to manage student expectations before introducing error prone representations.

4.3.4 Non-Representational Difficulties

Participants were from different schools, each with benefits and limitations. It became clear that the choice of educational tools to use, and the efficacy of those tools, was heavily affected by non-representational factors.

Infrastructure Problems

Issues relating to infrastructure constraints were discussed the most frequently, with teachers referring to issues relating to operating system choice, Internet connection speed, and computer specification. Two teachers, Keith and Mark, mentioned rejecting education tools based on system performance: "*We have thin clients but you can't use gamemaker or things like turtle. ...I'm hoping they are going to be replaced.*" and "*I was quite tempted by Unity. ...I toyed with it a bit, using it on some of the machines, but it was resource heavy*". Teachers and software developers also referred to issues using cloud-based tools with slow Internet connections:

> *"AppInventor - crashes, takes too long. All the files are held elsewhere. If you're trying to do controlled assessment then those files need to be held on school grounds. We had so many problems downloading the files online, it kept stalling and wasn't even suitable for an hour's workshop" (Andrea)*
>
> *"The issue with the online one is basically that you need a good Internet connection. ... Even the Scratch website doesn't render properly. You can't even read the buttons. So now we know that this blank button is the one you need to log in, but that's not really workable." (Margaret)*

In addition to dismissal based on infrastructure issues, it was clear that some tools were championed on the basis of compatibility: Mary said "So far it's working well. Key things are that it works on the network" and Stephen said "*The advantage to the .NET [Gadgeteer] is* *that everything's there in the box - we've got the power, the motherboard, the cameras and the joysticks, and everything there, and the touchscreen.*"

Lack of Supporting Knowledge and New Contexts

Both software engineers and four of eight teachers Stephen, Keith, Mary and Brian, stated that the choice of tool, and student attainment, were dependent on the level of students' supporting knowledge. This knowledge included mathematics knowledge, required to understand coordinates, degree angles, trigonometry, negative numbers and logic:

> *"...so I said ok, what happens when you multiply 5 by -1 and he couldn't answer by that point, maybe if he'd gone home and thought about it. And I asked, ok have you done any negative numbers, they had but possibly not with multiplication." (Margaret)*

Stephen found that although students knew the mathematics concepts, they found transferring them to a new context difficult:

> *"We do things like LOGO, they're fine with the basics, but the second you introduce aspects from different subject - degrees angles, they can't remember it. It's as if "it's not maths" so they can't get the fact it's a right degree angle as a right turn." (Stephen)*

In addition to mathematics knowledge, Margaret said that some students lacked written competency:

> *"Well there's one [student] who can't spell. So these projects, often they have like, ok and you need a variable and lets [sic] call it such and such. Then he'll call...sometimes he asks me to type it in. ...For scratch that's ok, for python, I don't think I will take that student to python." (Margaret)*

Documentation and Error Messages

When comparing visual and text-based programming environments, Mark noted that the documentation available for tools used in the classroom could be difficult for students to use, depending on the target audience: "[with Greenfoot, the] d*ocumentation is very programmer friendly, not very learner friendly. So the better documentation for the text based tools would be very useful"* and "*I think Scratch documentation is very good, much better than some languages, because it's written for kids that are learning rather than programmers*". Two teachers, Brian and Richard, mentioned that students did not do well with documentation:

> *"If there's a tutorial for them to work through, they don't act well, even with the top set, with lots of text in front of them. ...You might get half that could do well with a textual tutorial." (Brian)*

Mary noted that an alternative to documentation, and a useful way to keep students engaged, is to use video: "*Rather than go round I have video tutorials. Then you can have a class of 25 and they can be working at their own pace*". Both teachers and software developers mentioned error messages as a big source of confusion: Richard said "*You and I both know what 'Primary key should not contain a NULL value' means but your average primary school kid has no idea*" and Ben said "*...python's rubbish error generation. Even to me some of the errors are a bit arcane*".

Ability Dependent Concepts

All participants endeavoured to make concepts accessible to students with different levels of ability. Teachers used various strategies to avoid alienating lower ability students, such as avoiding representational difficulties like *Hidden Dependencies* *(CD)*: "*Every topic is accessible no matter what the ability range. ...The lower ability students we won't touch broadcasting with*" (Stephen), and avoiding representations with high E*rror Proneness* (CD) altogether: "*The lower ability students we stick purely with a WYSIWYG*" (Stephen).

Teaching Skill

Three teachers and both software engineers highlighted the difficult transition for teachers moving from a background in Information Communication Technology (ICT) to Computing: Andrea said "*I feel like I have just walked into a subject I haven't taught before*", while others noted that their colleagues had difficulties: Stephen said "*A lot of ICT teachers are non-computing specialists. [CSUnplugged] doesn't make sense to them – they struggle with how to explain it, and don't have the background*". A software engineer, Margaret, noted that "*None of the teachers from this school have taught any programing, and I don't think any of them know any*".

In sum, there are many non-representational factors that prevent teachers from selecting certain types of educational software. Such factors can also slow students' progression if not supported. Students find it hard to apply existing knowledge in new contexts, and sometimes do not have supporting knowledge. Support features like documentation can be beneficial in supporting learning for some students, but can be inadequate for other students.

4.3.5 Motivation, Liveness and Gender Differences

All teachers stated that to maintain engagement in programming activities, students need some source of motivation. The most frequently mentioned motivator for students was creating games:

> *"Pacman is a really good motivator. It's about making games, tricking them into writing a program. ...from that point on they are engaged in the activity". (Keith)*
>
> *"You can give them a simple Mario game… just having them instantly engaged tends to bring them back onside". (Stephen)*

Teachers noted that the reason games provide motivation is that students can build working prototypes very quickly: Mark said "*With games, getting a result is quite a short learning time*" and Brian said "*[Scratch makes it] easy to get immediate results, it's doing what they want". Heather went on to state "It's the instant gratification so they don't have to spend 10 minutes setting up a script*".

Web development was the second most frequently mentioned motivator for students. One teacher, Keith, highlighted websites as a big motivation for girls in particular: "*When you're doing HTML girls are really into it. When you're talking about the web they are really excited about it*".

Teachers highlighted differences between genders when using new programming tools, particularly in student motivation, confidence and tinkering. Brian observed that boys had more motivation and excitement about new tools, using them at home or in their own time. Stephen observed that the gender differences in motivation were more apparent in earlier years:

> *"In the younger years, it's very much a male side of things. Year 7s I seem to find the lads are more engaged. It tends to be because they've already done something, or dad has shown them something. We've got a few year 7 lads that can program well, possibly better than me. ...By year 9, the girls are by far the more engaged, more desperate to do well". (Stephen)*

Richard noted that boys have higher confidence than girls in general, and are keen start using new tools immediately, without looking at documentation. Mary, a teacher from an all-girls school noted that girls prefer more focused tasks and tend not to tinker much:

> *"90% of the room will say, "What are we doing? What do you want us to do? What do we have to do?" You will get a few that will play. ...My experience of girls is that they like a more focused task."(Mary)*

In sum, it appears there may be differences in motivation, confidence and tinkering between genders, particularly in Year 7 and Year 8 (ages 10-12), where boys display more motivation, high levels of confidence and tinkering. Teachers report that girls' confidence and motivation increases in subsequent years, but that this may be due to extrinsic motivation.

4.3.6 Teaching Approach and Assessment

Three teachers noted that they introduced students to new tools *without* emphasising any specific learning goals. They would then introduce concepts and theory later.

> *"I'm not teaching it from a programming point of view. We have worksheets and let them make games. Purely making games. ...Instilling the practice of doing, and why we are doing it later" (Heather)*

This approach is comparable to the constructionist approach advocated by Papert in Mindstorms (Papert, 1980). Ben noted that this approach was enhanced when the system gave quick feedback: "It's visual, it's immediate. …The kids get a buzz off that at the start".

Three teachers (Richard, Andrea and Brian) and both software developers stated that they had or were intending to use pair programming to support and motivate students. Andrea stated "*We try to get them to work collaboratively, in pairs, to share their ideas, get them to talk about what they have learnt*". Ben noted, "*We encourage them to swap around. Sometimes they are both driving it because there's a touchpad and a mouse…if they are in good pairs, and even, it works really well*".

When asked about assessment methods, three teachers (Richard, Mary and Stephen) stated that they used summative assessments after a fixed period working on a programming task. The assessment types ranged from an open classroom discussion on each student's work, to written assessments where students were asked to reproduce programs on paper:

> *"I found that the students become slightly dependent on pre-made things. They don't want to think about "what do I need to write" to get this to search for a location, they just want to fit the jigsaw piece. Even to the point where you can turn around and test the student at the end of the Scratch topic, and ask them to write how you start a program. They cannot remember that it's red and green*

flag click, and yet they have used it for 8 weeks, and made these amazing programs, and they can't remember the actual codes" (Stephen)

Stephen also stated that assessment of conceptual understanding depended on the content being taught:

> *"Depends on the concept. We give them questions at the end if it's theory – past paper questions it tends to be. ... the idea of introducing an integer – I'll do it in one lesson. Database connections, I'll do it over a few lessons. You assess towards the end as you go along". (Stephen)*

In sum, teachers introduce students to motivating educational tools and encourage constructivist approaches to learning. As students become familiar with the environments, concepts are introduced in the context of the construction goal. Students' practical abilities are assessed at the end of a predefined period of time based on the final code and functionality.

4.3.7 Demonstration Feedback

The final portion of each interview was devoted to demonstrating a software prototype of bi-directional translation between text code and Direct Manipulation. In contrast to the semi-structured nature of previous questions, participants were asked direct questions about the prototype (see Appendix A).

When asked what part of the demonstration would be most valuable to student education, all teachers responded positively: Heather said "To be able to see that going from code…to see there's an underlying code for something, yeah that would be good",

> *"If you altered the circle, and they could see the value at the same time. It's that sort of thing. As you change one thing you can see it in the code and it gives you more of an idea what the code represents". (Brian)*

Teachers reacted positively to the liveness of the environment: Andrea said "*I like the fact you can experiment and see quickly rather than having to go through save it and view it on a browser. Really quick instantaneous option*", and Keith said "*As long as it works in the way kids would imagine it. ...It would encourage tinkering definitely.*"

Participants were also asked to comment on the importance of candidate features for the environment, including collaboration features, automatic evaluation, helper agents, integrated tutorials, and the ability to continue at home. Participants felt most strongly about the ability for students to continue using the software at home: Andrea said "*Vital – especially if they are* *completing homework tasks*", with one participant mentioning that a cross platform solution would be most suitable: Mark said "*From the school's point of view, you always set a piece of homework that requires something, and you always get 'I haven't got a computer'. Something that would run cross platforms would be amazing*". Teachers generally responded to other candidate features with nervousness, stating that automatic evaluation and integrated helper agents could result in demotivation, and highly collaborative systems could result in distraction.

Finally, teachers were asked what additions would be required to use the prototype system in a real classroom; responses including tooltips, useful error messages and configurable code completion. Keith stated that a limited amount of documentation might be useful: "*I think with any system, people are put off with something that requires a lot of documentation. If you're able to right click on something and have a very brief intro and people can roll with it, that would make it more popular*".

In sum, participants reacted positively to the prototype demonstration, suggesting the liveness of the environment would encourage tinkering and increase understanding of the text-based representation. The most requested feature was for the system to be usable from home, and to work cross-platform. Teachers emphasised the requirement for useful error messages and terse, educational documentation.

4.4 Conclusions

I have presented results from ten, two-part interview sessions, conducted with eight professional teachers and two software developers. I carried out thematic analysis of semi-structured interview data using pre-defined codes based on Psychology, HCI and Computer Science Education literature, in addition to emergent codes identified during transcript analysis.

Analysis of findings related to *a priori* codes and emergent codes from semi-structured interviews resulted in five major categories or themes: (1) linking concrete and abstract representations and (2) representation-specific difficulties, (3) non-representational difficulties, (4) motivation, liveness and gender differences, and (5) teaching approach and assessment.

Codes Elicited Directly from Questions

All teachers found the high error proneness of text-based representations to be a substantial barrier for students, and in some cases had to reassure students that encountering errors was normal. Teachers mentioned that students often lacked resilience and debugging skills. They made recommendations to suggest that any new tool should have suitable error messages and integrated documentation.

Teachers found that non-representational factors such as infrastructure constraints heavily influenced their choice of educational tools. They also stated that the lack of supporting knowledge in both students and teachers limited the extent to which either could experiment with new tools or features. Teachers recommended that a new tool should be usable by students from home, and cross platform to deal with infrastructure problems.

Finally, teachers described their assessment style, and alluded to differences in motivation and confidence between genders, particularly with younger students. They stressed that any new tool should aim to attract both genders, and encourage tinkering and experimentation.

Emergent Codes and Unexpected Responses

Emergent codes identified during analysis were added to the list of codes before themes were identified, in accordance with framework and template analysis. The main theme to emerge from corpus analysis – "Linking concrete and abstract representations", is composed of emergent codes based on teachers' deliberate introduction of education software tools to support programming progression throughout secondary school education. Teachers might typically start by using oral or paper-based descriptions of algorithms, and move on to using visual language tools that provide intrinsic motivation for students in the form of games or animation. In addition to the introduction of practical tools, teachers reported that they would introduce abstract concepts, such as algorithms or variables, using concepts or objects that students were already familiar with, such as making a cup of tea, or putting an object in a box.

Additional emergent codes included "teaching skill", in which three teachers and both software engineers described a current lack of teachers with the skills required to teach programming. This code was linked with the theme "Non-representational difficulties". Codes also emerged relating to teachers' descriptions of strategies used to increase student engagement in programming exercises, such as encouraging students to developing retro games or websites, and using tools that provide quick feedback. Finally, codes emerged to describe the use of teaching methods. For example, three teachers and both software developers described teaching methods such as pair programming, student collaboration (e.g. encouraging students to “ask three before me”) and the use of video tutorials.

The Design and Deployment of Programming Software in Schools

Responses from teachers and software developers during the interviews provided a basis on which to inform the design, implementation and deployment of a new programming education system. Interviews found that teachers currently use implicit representation transition strategies, but often do not attempt to emphasise correspondences between representations. New software must therefore attempt to make these links explicit to further support bridging between representations.

The most common difficulty teachers identified was the high error proneness of text-based environments such as Python or Greenfoot. Any text-based representation must therefore provide support for error highlighting and recovery. It may also be helpful for systems to provide code to scaffold student understanding of the syntax elements available before they are required to create their own programs. The second most common difficulty was the difficulty in viewing large programs, owing to reduced *visibility* (CD) and high *diffuseness* (CD) of visual language notation. The design of visual languages in new systems must take this into consideration to allow students to view a complete program without frustration.

Interview participants raised a number of non-representation related difficulties students experience when learning to program. The most common difficulties were infrastructure-related, and occurred when software required a specific operating system or speed of Internet connection. New systems must therefore be platform agnostic, and run without requiring an Internet connection. Other common difficulties included a lack of supporting knowledge. The design of any new system must therefore attempt to minimise the amount of supporting knowledge required, or provide mechanisms that allow students to easily infer how the program works.

Chapter 5 DrawBridge

5.1 Introduction

This chapter describes the use of design heuristics developed from the MoRA framework presented in Chapter 3, and findings from semi-structured teaching interviews discussed in Chapter 4, to create a new educational programming tool called DrawBridge, which supports students when making the transition from minimal-abstraction tangible programming representations to unlimited-abstraction symbolic programming representations. The chapter begins by giving a brief description of DrawBridge, followed by a rationale for key design decisions, their effect on the usability of the system, and a description of the way in which they contribute to educational objectives. The second half of the chapter describes practical implementation details and limitations of the system.

Descriptions of features added to the second iteration of DrawBridge, including integrated assessments, changes in the code style, improved syntax annotation, and the ability to load and save sessions, are discussed in Chapter 8.

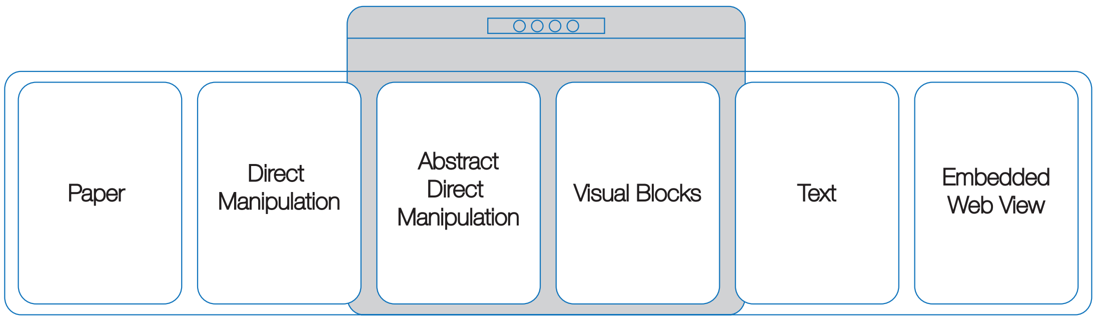


Figure 5.1: DrawBridge Overview

(DrawBridge shows two representations at once, side by side. Users can use navigation buttons to move left and right between screens using a panorama metaphor. On moving left or right, one representation leaves the view, and another enters it.)

5.2 DrawBridge

DrawBridge is a prototype MER environment consisting of 6 programming representations (see above), of which 2 can be viewed at the same time. DrawBridge was so named because before using the system, students **Draw** characters on paper. When they use the system, they are supported by **Bridging** features, which allow them to make the transition between programming representations.

Table 5.1: User Journey in DrawBridge

Step 1	Draw characters on paper.
Step 2	Take picture/scan paper.
Step 3	Open DrawBridge and import the image. Image is placed on “Paper” panel.
Step 4	System positions the segmented characters in “Direct Manipulation” Panel
Step 5	Use “Abstract Direct Manipulation” panel to record animations.
Step 6	Use blocks on “visual blocks” panel to modify animations.
Step 7	Add to text code on “text” panel to create new animation steps.
Step 8	View the HTML5/JavaScript output in DrawBridge or external browser.

The steps above describe the user journey in DrawBridge. A typical workflow might be for a student to first draw characters on paper, similar to those shown in Figure 5.2. DrawBridge allows the user to import a digital image taken of their characters, which is loaded and preserved on the first “paper” panel on the left side of the screen. The image is copied and segmented into smaller images of each character, which act as scaffolding objects on the “Direct Manipulation” and “Abstract Direct Manipulation” panels. Programming code to draw each character in the correct position is also generated on “Visual Blocks” and “Text” panels.

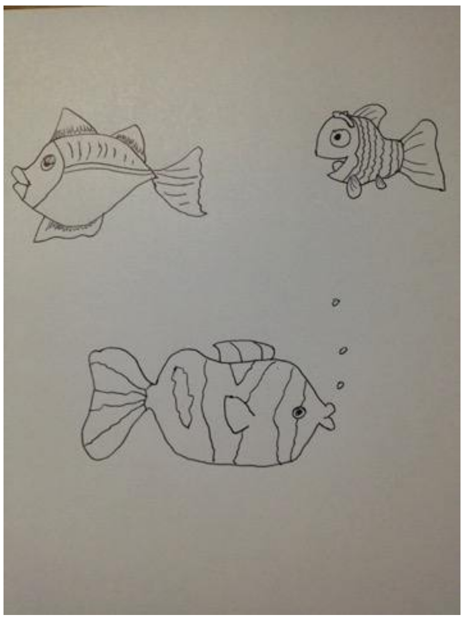
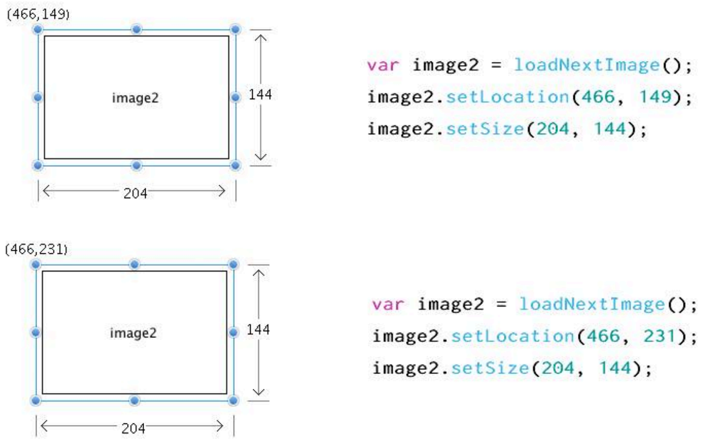


Figure 5.2: Example of a Photo of Characters used with DrawBridge

After image import has taken place, students can interact with the system by moving and resizing characters on either the DM panel, where they appear as bitmap images, or Abstract DM panels, where they appear as abstract rectangles that emphasise the image coordinates and size.

Users begin their journey by moving and resizing characters using the Direct Manipulation or Abstract Direct Manipulation panels, which are tightly bound to the programming code via an underlying model. Any change via direct manipulation results in a change to the model and therefore the existing programming code (see Figure 5.3).



```
var image2 = loadNextImage();
image2.setLocation(466, 149);
image2.setSize(204, 144);
```

```
var image2 = loadNextImage();
image2.setLocation(466, 231);
image2.setSize(204, 144);
```

Figure 5.3: Bi-directional binding between Abstract Direct Manipulation objects and Programming Code

When users record animations with their characters in the Abstract DM panel, the tight binding of characters and code is broken; instead of being modified, the existing code is appended with new code for each animation step. Before animating, students are able to edit code in Direct Manipulation, Visual Block and Text representations in a bi-directional manner, allowing them to use the most appropriate representation for the task they want to complete.

5.3 Design Rationale

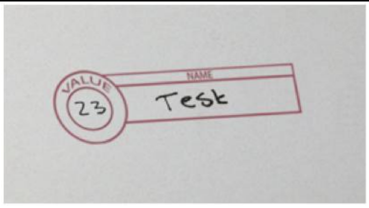
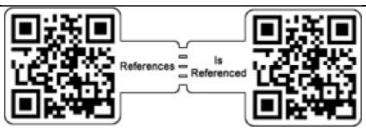
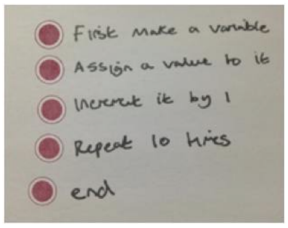
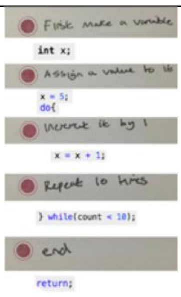
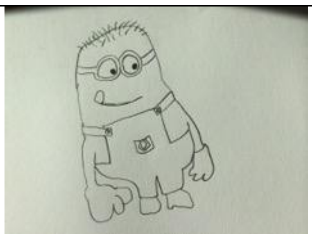
The MoRA framework presented in Chapter 3 suggests that student acquisition of notational expertise (NE) could be improved by using effective representation transition strategies. One such strategy is to allow students to move from concrete tangible representations to abstract symbolic representations by increasing abstraction while improving familiarity with each type of representation (tangible, graphic or symbolic).

5.3.1 Applying Heuristics from the MoRA Framework

Design heuristics created in the MoRA framework suggested that the system could use five representations to allow students to make the transition from tangible minimal abstraction representations to symbolic unlimited abstraction representations. Intermediary representations were designed after the first and final representations, so that they could be developed to appropriately support smooth transitions between representations, and help students to develop the appropriate mental operations required to use the later representations.

Many concrete tangible representations were considered as a starting point for DrawBridge (Table 5.2), including using rubber stamps to specify the initial state of a program, supplying Tear-off QR codes to create references between objects, and allowing students to draw characters that would be used as objects-to-think-with. Each proposed representation however, requires mental operations that the student may not have developed, such as how to specify algorithms and procedures, and problem decomposition.

Table 5.2: Candidates for First Tangible Representation

Method Name	**Requisite Mental Operations**	**Illustration**	**Recognition Equivalent**
Rubber Stamps	Algorithms & Procedures (Assignment, types)		var Test = 23;
QR Tear-off References	Algorithms & procedures (Reference)		var x = y;
Stamp-bullets	Problem Decomposition / Algorithms & Procedures	 First make a variable Assign a value to it Increment it by 1 Repeat 10 times end	 First make a variable Assign a value to it Increment it by 1 Repeat 10 times end `int x;` `x = 5;` `do{` `x = x + 1;` `} while(count < 10);` `return;`
Character Capture	Data Collection		

The "character capture" method, inspired by the DENIM system (Lin, Newman, Hong, & Landay, 2001), was selected as the most suitable use of the first tangible representation as it required the least prior knowledge and the fewest pre-requisite mental operations. The process of drawing using pencil and paper was expected to be familiar to students from school art lessons and early childhood, and therefore provide little or no barrier to entry for students.

Further, this method was expected to give students an increased sense of agency within the resulting program, helping to reify abstract representations, and providing a source of motivation. I suggest that hand-drawn characters may be even more motivating than existing educational programming objects, such as the cat in Scratch, or the turtle in LOGO, due to their unique and personalised nature.

5.3.2 Programming Semantics

The semantics of a programming language defines the process used to evaluate the commands and structure. A change in semantics could the change the behaviour of program execution. For example, the statement 'x = y' in a language with imperative semantics may mean that x takes the value of y. In a declarative semantics, it may mean that y is tightly bound to x, resulting in a change in the value of y on a change in the value of x. A potential barrier to success when using separate educational programming systems is that students may have to learn a new set of semantics for each system they use, making transitions between systems more difficult. Integrated MERs can address this by allowing students to learn one underlying set of programming semantics for all representations, reducing the effort to make the transition between each representation.

The choice of which semantics to use is highly dependent on the selected text language, which in this system will be positioned as the final representation. The final choice of text language – JavaScript – was influenced by the following factors: (a) feedback from teaching interviews revealed that Python, JavaScript and SmallBASIC are often already used in the classroom to introduce text languages; (b) JavaScript requires minimal changes to existing school infrastructure, which teachers identified as a high priority; (c) analysis of worldwide programming language use suggest that JavaScript has become the most popular language (O'Grady, 2015); (d) JavaScript is the most in-demand skill in professional contexts (Joseph & Siganakis, 2014); (e) JavaScript borrows most of its syntax from Java (Eich, 1996), which is derived from C and C++, making it similar to at least 3 other languages in the top 10 most popular in the world, (f) for those reasons, teachers are likely to be familiar with JavaScript syntax; (g) JavaScript has a very high level of online support; (h) open source code exists to parse and render JavaScript, reducing engineering effort.

5.3.3 A Cognitive Dimensions of Notations Analysis

As discussed in Chapter 2, the degree to which representations are more or less suitable for particular types of task is dependent on properties of the representation. The Cognitive Dimensions (CDs) of Notations framework contains a list of orthogonal dimensions that describe such properties in the context of programming. These dimensions act as useful tools to pair, compare and evaluate representations, and were used during the design of DrawBridge to choose appropriate representations that correspond with the transition strategies presented in Chapter 3.

Table 5.3: Mapping from MoRA Strategy to DrawBridge Representations

MoRA Classification	Representation
Tangible MARS	Characters on Paper
Tangible LARS	Direct Manipulation
Graphic LARS	Abstract Direct Manipulation
Symbolic LARS	Visual Block Language
Symbolic UARS	JavaScript Text

5.3.4 MER Design by Analysis of Representational Trade-offs

Representation 1: Drawing Characters using Physical Pen and Paper

Paper has well-understood usability characteristics (Sellen & Harper, 2003). Although it has the flexibility to support all three of Bruner's modes of representation: enactive, graphic or symbolic, paper has particular advantages that make it a suitable first representation in this system.

Paper might be described as abstraction-tolerant, meaning it can be used as it comes, but can also be used to create new abstractions (Green & Petre, 1996a). If students' use of paper were limited to creating simple 2D characters, the level of abstraction encountered would likely be minimal, resulting in a representation with a low a*bstraction gradient* *(CD)* and therefore reduced requirements for students starting to learn with DrawBridge.

The *closeness of mapping* *(CD)* of a programming notation is used to describe the distance between the program world, and the real world. In the proposed paper representation, the characters would be the real world objects referred to by the program. It follows that characters drawn on paper have a *high closeness of mapping* *(CD)*. Similarly, the role of each character is likely to be obvious to its creator, resulting in high *role expressiveness* *(CD)*.

A major benefit of drawing characters on paper is that students can easily view the current state their work. In the CDs framework, the ability to evaluate the current state of the system is referred to as *progressive evaluation* $^{(CD)}$. As students are drawing their characters on paper, they can pause to reflect on their progress, and plan further modifications or additions as a result of their reflection. Additionally, characters drawn on paper are likely to have *high visibility* $^{(CD)}$ and *juxtaposability* $^{(CD)}$, as the content is limited to an area of paper that can be scanned or captured using a camera.

The advantages of drawing characters on paper are counterbalanced by several disadvantages; for example, students may not be able to write notes or annotations (*secondary notation* $^{(CD)}$) next to their characters, as the software would interpret them as additional characters. However, secondary notation is primarily used to aid readability, and is therefore unlikely to be required in this context. Pen on paper is usually highly *viscous* $^{(CD)}$, making it difficult for students to edit content that has already been created. This disadvantage is alleviated by the fact that student can start afresh on new paper if significant changes are required.

Representation 2: Direct Manipulation of Characters on Screen

The second proposed representation is computer based, and generated from a photograph or scan of characters drawn on paper. If individual characters could be robustly segmented from the image, their properties could be directly manipulated to the satisfaction of the student, providing further opportunity for reflection, and the chance to address issues resulting from the highly viscous paper representation. For example, objects could be repositioned, resized, rotated, skewed or distorted. Further benefits of Direct Manipulation, such as presenting the user with continuous representations, allowing them to use physical actions to interact with digital objects, allowing incremental reversible operations whose impact on the object is immediately visible, and requiring minimal knowledge to get started, are well established in HCI (Shneiderman, 1981).

When evaluating the suitability of Direct Manipulation as a representation, one must consider the properties that can be manipulated, and the interaction required to manipulate them. Like paper, the Direct Manipulation (DM) representation is likely to have a low *abstraction gradient* $^{(CD)}$ and high *closeness of mapping* $^{(CD)}$ as objects would be recognisable as the same object the user had drawn on paper. Unlike paper, DM supports low *viscosity* $^{(CD)}$ and therefore high *juxtaposability* $^{(CD)}$. Each object can be treated by the interface as a character, and could therefore be manipulated in the same way, resulting in high *consistency* $^{(CD)}$. The high *visibility* $^{(CD)}$, reversibility and incremental nature provided by DM interfaces are also likely to result in low *error proneness* $^{(CD)}$ and few *hard mental operations* $^{(CD)}$ for students.

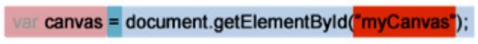
Representation 3: Abstract Direct Manipulation and Programming by Demonstration

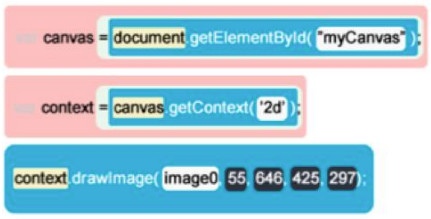
The third proposed representation acts as extension of the aforementioned DM interface by providing the benefits of DM, but increasing the abstraction level. It would do this by increasing the *closeness of mapping* $^{(CD)}$ and reducing the *role expressiveness* $^{(CD)}$. Instead of using a DM object that resembles the character drawn on paper, this representation could use a DM object that emphasises the shared properties of characters, such as their position and size, which would allow the student to build an abstract model of characters, their properties, and how they might be used or stored in a computer. The emphasis on these properties may even allow students to reflect on how they might go about changing them within the program.

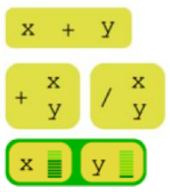
The DM representation is particularly suitable for creating new programs via Programming by Demonstration (PBD), which allows users to apply changes to properties of objects in a system while the system observes the user's actions (Cypher, 1993). Once all actions have been observed, the program can be executed as many times as is required. Users can think of this representation as allowing them to "record" their program, and play back their recording. The result of paper segmentation and PbD can be used to automatically create scaffolding code in other representations, which can be viewed and modified by the user. This idea follows the view-modify-create cycle, recommended in a workshop on Computational Thinking (The National Academy of Sciences, 2011).

Representation 4: Visual Blocks

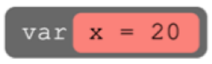
The fourth proposed representation is a novel visual block language, which is able to exploit properties found in existing visual languages, while easing students towards a transition to text representations. The proposed visual block representation would allow each block to be positioned using drag and drop, resulting in low *viscosity* $^{(CD)}$.


```
var canvas = document.getElementById("myCanvas");
```
(a) Experimental Low-diffuseness Visual Block Design


```
canvas = document.getElementById( "myCanvas" );
context = canvas.getContext( '2d' );
context.drawImage( image0, 55, 646, 425, 297);
```
(b) Experimental Nested Visual Blocks



(c) Experimental mathematics-style blocks with value visualisation



(d) Experimental Nested Structured Editor

Figure 5.4: Prototype designs for Visual Block representations

Although block languages usually use more space (*high diffuseness* $^{(CD)}$) than text representations, Figure 5.4-a shows that this does not have to be the case. However, increased *diffuseness* $^{(CD)}$ allows users to target and move blocks easily, reducing *viscosity* $^{(CD)}$, and emphasising program structure, increasing *visibility* $^{(CD)}$. Further, clear block colouring according to type improves *role expressiveness* $^{(CD)}$ and *consistency* $^{(CD)}$ of the representation. This emphasis on structure and reduced viscosity that allows students to tinker with their program is likely to be essential, particular for novices who may never have seen a program before.

Existing block languages such as Alice and Scratch achieve low *error proneness* $^{(CD)}$ by requiring that blocks are fitted together like LEGO, or puzzle pieces (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). While this mechanism protects novices from errors, which may reduce motivation and confidence and ultimately self-efficacy, it also masks some difficulties inherent in programming, and may simply delay the requirement for debugging skills until students begin using text-based languages, where they are also expected to overcome other notational difficulties. The proposed visual blocks then, are structured more like text, which allows students to begin acquiring debugging skill using a low-viscosity representation before moving to text.

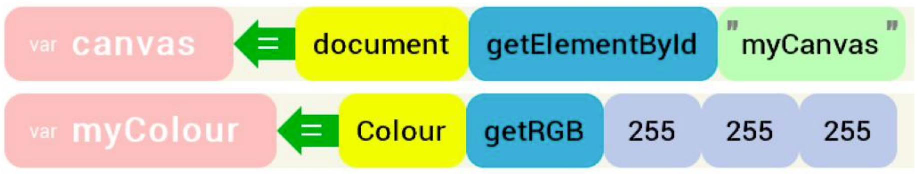


Figure 5.5: Final Design of Visual Blocks

Representation 5: JavaScript Text

The fifth proposed representation is an unlimited abstraction text editor, which provides students with a realistic experience of editing programming text in a conventional programming environment.

Text programming representations typically only require a small amount of space to express complex commands. This low *diffuseness*$^{(CD)}$ can be beneficial for expert users, who value terseness and precision, but could overwhelm novices by increasing cognitive load and the number of *hard mental operations*$^{(CD)}$ required. The introduction of abstract symbols also reduces the *closeness of mapping*$^{(CD)}$ between the program world and real world, and the *role expressiveness*$^{(CD)}$ of each part of the notation. The *viscosity* $^{(CD)}$ of the representation is likely to be highly user-dependent; novices may be required to make many key presses or use their mouse to direct manipulate the text, making it highly viscous, whereas experts may know shortcuts to speed up modifications, making it less viscous. For example a novice might manually edit each instance of a variable when changing its name, while an expert might use regular expressions in find/replace.

The proposed text representation improves *role expressiveness*$^{(CD)}$ and *visibility*$^{(CD)}$ of standard text by using syntax highlighting to allow users to identify structure, and error annotation to draw attention to errors in the code. Although syntax highlighting may not provide as much structural emphasis as visual block languages, it has still been shown to be an improvement over raw text in search tasks (Baecker, 1988). In addition to syntax and error highlighting, this text representation includes features to highlight text that is currently being edited in other representations, explicating otherwise implicit parts of the text (e.g. the meaning of parameters), and further emphasising the relationships between representations.

5.3.5 Representation progression

A key design decision when creating DrawBridge was the method by which students should make transitions between representations. MERs described in previous chapters, such as Leogo, allow users to view several representations at the same time. Although this may be useful for expert users, the amount of new information on the screen could unnecessarily increase users' cognitive load and therefore reduce confidence by increasing task complexity, mental load and mental effort (Kirschner, 2002). Furthermore, the way in which students make the transition between representations could not be controlled, making it inconsistent, and therefore difficult for teachers to support. From a usability perspective, any increase in the number of representations that appear on a fixed size screen reduces the space available for each representation, reducing *visibility* $^{(CD)}$ and increasing the number of *hard mental operations* $^{(CD)}$ required for searching tasks.

The proposed method for presenting representations in DrawBridge is to display two on the screen at once. This configuration limits the amount of new information a student is required to process, and allows the system to support correspondences between the representations. Further, when making the transition to new representations, students should always be able to see at least one representation they are already familiar with.

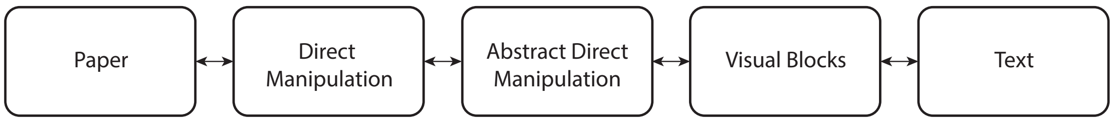


Figure 5.6: Representation Transition

There are several possible mechanisms for changing the representations that appear on the screen. The two most promising methods are shown in Figure 5.7. The first allows one representation to be "pinned" into position, while others would move vertically beside it. In this configuration students would be able to view correspondences between the fixed representation and every other representation, but would not see correspondences between the vertically moving representations.

The second transition mechanism, which became the final design, uses a panorama metaphor, in which the screen appears to move across a larger sequential arrangement of representations. This configuration requires that students follow a fixed path when making the transition between representations, and ensures that each transition can be tailored to support knowledge obtained in previous representations. In both designs, edges of the previous and next representation can be shown to make navigation options clear to the user.

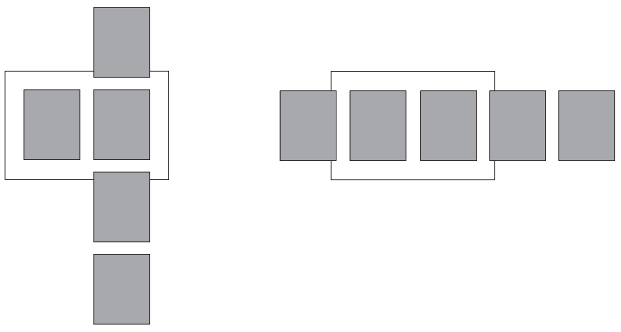


Figure 5.7: Representation Transition Movements

Left: One representation is fixed, with others moving up and down relative to the screen **Right**: Representations move left and right relative to the screen.

Maximising the Usability of Paired Representations with Cognitive Dimensions

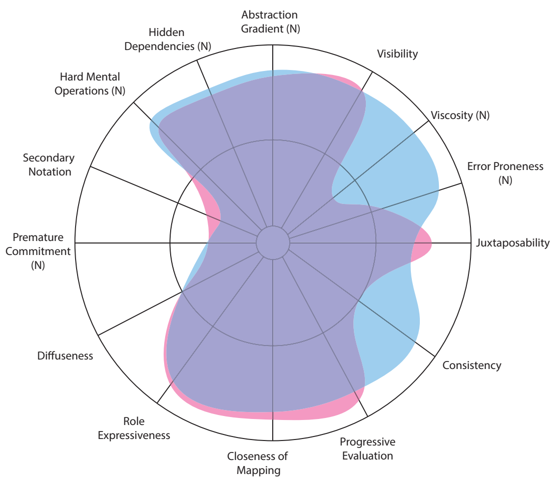
The proposed configuration of a sequential, two-representation display, allowed the usability of each pair of representations to be evaluated for incrementation, modification and exploratory design tasks using the CDs framework. As users make the transition from low-abstraction pairs of representations to high abstraction symbolic representation pairs, the combined usability of each pair decreases. This decrease is required as it introduces valuable learning opportunities for students that allow them to ultimately use symbolic representations in a way that is external valid. The speed at which users make the transition can be controlled by either the user or their teacher, and transition mechanisms described above enable the user to move back and forth between pairs of representations at any time. The bi-directional nature of the system also allows users to use representations they are comfortable with until they decide to make the transition to symbolic representations.

The first pair of representations, shown in Figure 5.8-a, consists of paper and Direct Manipulation (DM). The superimposed view of the usability characteristics for both representations shows that they have similar properties, but that DM is less *viscous* (CD)*,* as objects can be moved around easily, less *error prone* (CD), as the user is limited to reversibly moving and resizing objects, and more *consistent* (CD) than paper, as each object is treated in the same way. The addition of DM therefore complements the paper representation, and allows them to bypass trade-offs made when just using paper, such as high *viscosity* (CD). This combination of representations results in a pairing that has high premature commitment, and a low provision for secondary notation. However, other dimensions have favourable levels of usability for novice programmers, making the pairing suitable for the first representations used in the system.

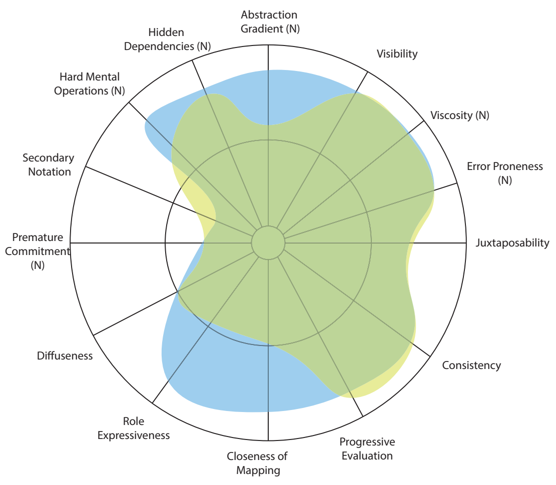
The second pair of representations, shown in Figure 5.8-b, is DM and abstract DM. Unlike the previous pair, this pairing moves to a target representation (abstract DM) with less desirable usability characteristics. Abstract DM has increased hard *mental operations* (CD) due to the difficulty in identifying which rectangle corresponds to which character, a higher *abstraction gradient* (CD) due to each object using the same visual appearance, and reduced *closeness of mapping* (CD) and *role expressiveness* (CD), due to lack of visual correspondence with the original characters. However, these trade-offs must be made in order to increase students' acquisition of Notational Expertise so that they can better cope with later representations. Despite moving to a representation with reduced usability characteristics, the combined usability characteristics of this pairing are very similar to the previous pairing.

The third pair, shown in Figure 5.8-c, compares the usability characteristics of abstract DM to visual blocks. The target representation, visual blocks, has increased *hidden dependencies* $^{(CD)}$, as it involves variables and functions that can refer to each other, *hard mental operations* $^{(CD)}$, as it introduces concepts of variable declaration, assignment and functions. The visual block representation also has increased *viscosity* $^{(CD)}$, as when a change is made, each block needs to be positioned in a way that is valid. In some cases, it may be easier for users to modify object properties using DM, rather than visual blocks. This pairing introduces usability challenges when compared to the previous pairing, such as reduced *closeness of mapping* $^{(CD)}$, and increased *abstraction gradient* $^{(CD)}$.

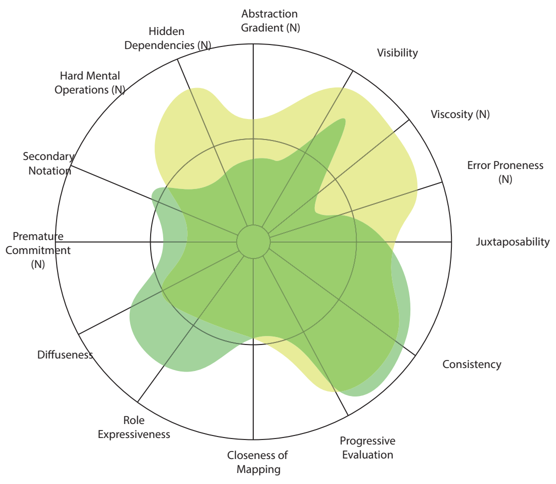
The final pair, shown in Figure 5.8-d, consists of visual-block and text representations. It is worth highlighting here that the usability of text representations according to the CD framework is relatively low for incrementation and exploratory design tasks, making it an undesirable first representation. However, in DrawBridge, the addition of a visual block representation is intended to reduce *viscosity* $^{(CD)}$ by supporting drag and drop of blocks, increase *role expressiveness* $^{(CD)}$ by emphasising structure with colour, and therefore reduce *hard mental operations* $^{(CD)}$. The reduced *diffuseness* $^{(CD)}$ of text makes it highly *visible* $^{(CD)}$ and *juxtaposable* $^{(CD)}$. This pairing has many usability challenges for novice programmers when compared to the previous pairing, such as an increased *abstraction gradient* $^{(CD)}$, increased *hidden dependencies* $^{(CD)}$ and high *error proneness* $^{(CD)}$, making it best suited to be the final representation pairing.



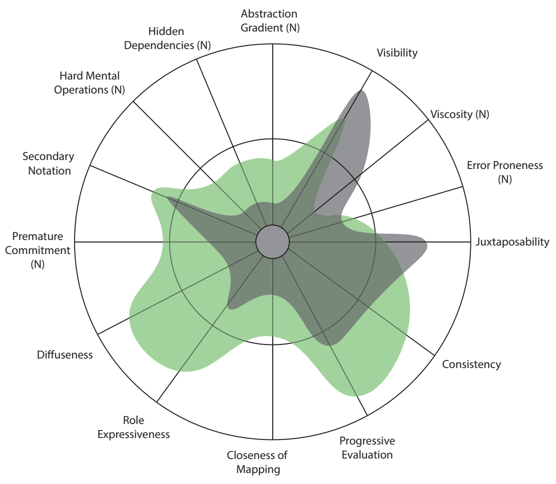
(a) Paper (Pink) and Direct Manipulation (Blue)



(b): Direct Manipulation (Blue) and Abstract Direct Manipulation (Yellow)



(c) Abstract Direct Manipulation (Yellow) and Visual Blocks (Green)



(d) Visual Blocks (Green) and Text (Black)

Figure 5.8: Cognitive Dimension Comparison Between Representation Pairs

CDs marked (N) are negatively rated (e.g. a high rating for Viscosity (N) means low viscosity).

Reaching Educational Goals with Each Representation

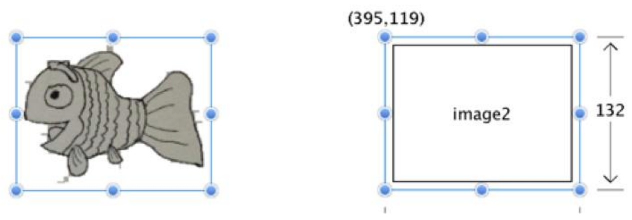
In addition to providing usability benefits, each representation, and pair of representations, is designed to help students build new mental operations, and develop a conceptual model of the underlying programming semantics in DrawBridge.

The transition from **Paper** to **Direct Manipulation** demonstrates that physical objects in the real world can be represented or modelled digitally via some transformation. The transition from **DM** to **Abstract DM** shows how images are stored and represented by a computer (i.e. a rectangular block of pixels with a location on the screen, and size), and how the properties of objects can be used to create abstractions. The transition from **Abstract DM** to **Visual Blocks** shows users how programs can be used to specify the properties of images drawn on screen, how simple commands are constructed, and what the semantics of the underlying language are. The transition from **Visual Blocks** to **Text** code further emphasises the underlying program semantics, but also allows the user to build knowledge of the textual syntactic elements (e.g. semi-colon, brackets) used to structure text programs.

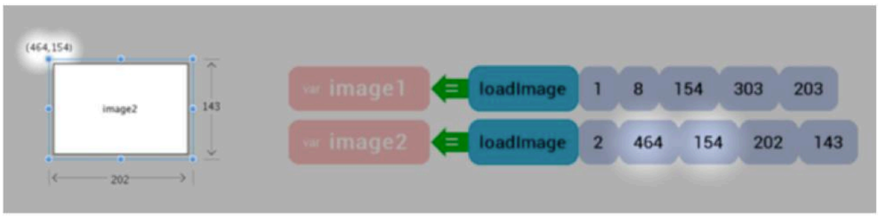
Highlighting Correspondences between Representations

The proposed sequential two-representation configuration allows DrawBridge to explicitly highlight correspondences between each representation, which are necessary to improve students' notational expertise. More specifically, correspondence highlighting could improve students' ability to recognise ideas embedded within a particular representation, identify correspondences between representations and translate from one representation to another.

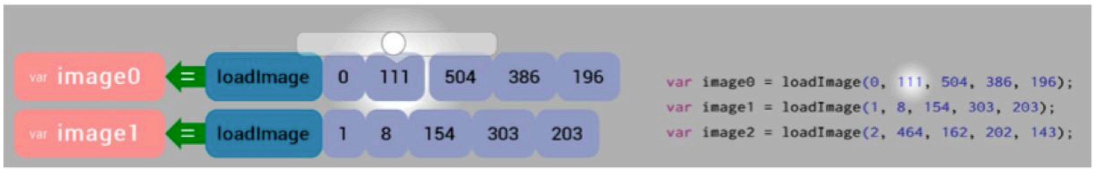
Correspondence highlighting does not always provide clear benefits to users (Nevins, 2009), and may only provide significant value when used to support complex representations. Highlighting in DrawBridge must therefore be highly visible while preserving the usability of the existing representations. Figure 5.9 shows three instances of correspondence highlighting between interactive pairs of representations in DrawBridge. In both DM panels, selection of an object is indicated using a bounding box; when an object on one DM panel is selected, the corresponding object on the other panel is also selected. Correspondence highlighting in visual block and text representations darkens the background of each representation, and highlights the part of the representation that is changing using a radial, Gaussian gradient.



(a) Correspondence highlighting between DM (left) and abstract DM (right).



(b) Correspondence Highlighting between Abstract DM (Left) and Visual Blocks (Right)



```
var image0 = loadImage(0, 111, 504, 386, 196);
var image1 = loadImage(1, 8, 154, 303, 203);
var image2 = loadImage(2, 464, 162, 202, 143);
```

(c) Correspondence Highlighting between Visual Blocks (Left) and Text (Right).

Figure 5.9: Correspondence Highlighting

5.3.6 Motivation

Interviews with teachers in Chapter 4 found that students were greatly motivated to program when using their programs to create games. The success of Scratch, and interest in newer programming environments based around games (Esper, Foster, & Griswold, 2013) (H. Collins & Target, 2013) also supports this observation.

Although games are motivating, they provide distraction from the main goal of using DrawBridge, which is to develop notational competence. With this in mind, DrawBridge was designed to support simple animations, similar to those found in key-frame animation, in which objects move from one "key" position to another over a given time or number of frames. In DrawBridge, students can create simple animations by "recording" their drag and drop actions on the Abstract Direct Manipulation panel. Recording the animation is a form of Programming by Demonstration (PbD), and allows the system to generate programming code, which appears in the Visual Block and Text panels.

5.3.7 Liveness

When students begin to use a new programming language, they are likely to benefit from high *progressive evaluation* $^{(CD)}$, or the ability to check whether the program they have specified carries out its intended function. Progressive evaluation can allow parts of the representation to act as pseudo-documentation to display the effect of new commands. It can also help to reinforce, controvert or extend students' mental models.

In programming environments, the term for decreasing latency between actions performed by the user, and the execution of the program to show their effects is referred to as "liveness". Tanimoto defines up to six levels of liveness in programming environments, ranging from level-1 – *informative*, where the representation cannot be executed at all, all the way to level-6 – *strategically predictive*, where the environment displays several executing variants inferred from the current program (S. L. Tanimoto, 1990; S. Tanimoto, 2013). Existing educational programming environments have employed liveness features categorised at level-2 (Maloney et al., 2010), allowing users to execute portions of code on demand, level-3 (Hundhausen & Brown, 2007), allowing users to trigger execution via edits to the program, and level-4 (Aaron & Blackwell, 2013), where the program perpetually executes while users hot-swap new functions into memory.

The proposed system will complement existing work by employing Tanimoto's liveness level-3, allowing students to view updates to the program as they make edits, and level-4, allowing program execution to be looped so that the results of edits appear on the next loop. Further increases in the level of liveness could reduce opportunity for student reflection (Beckwith et al., 2006; O'Hara & Payne, 1999), and would require significant technical effort to implement. However, exploring increased liveness in educational programming environments may serve as an interesting future direction.

5.4 Implementation

DrawBridge was written in Java for several reasons. Firstly, Java programs are portable and cross-platform, and therefore are not tied to any specific operating system or existing software (other than the Java Runtime Environment (JRE), which is installed on 89% of computers (Oracle, n.d.)). Secondly, Java supports multithreading, and native graphics rendering, which is required by the application. This functionality is only now becoming fully supported in web browsers.

The DrawBridge codebase consists of approximately 18,000 lines of Java, not including external libraries. Google's Caja project was used for JavaScript lexing and parsing, and a custom JavaScript Abstract Syntax Tree (AST) and interpreter were implemented to support bi-directional translation between representations. The system was instrumented to record user interactions by sending all interaction data to a central server.

5.4.1 Architecture

An overview of the core architecture of DrawBridge is presented in Figure 5.10. Each panel implements a ParserListener interface, which allows them to receive updates when the AST is changed or recreated. The JSInterpreter is a custom build interpreter that allows users' code to be executed within the environment. Figure 5.11, Figure 5.12 and Figure 5.13 show a full screen view of each distinct pair of representations in DrawBridge. The three pairs are shown together in Figure 5.14.

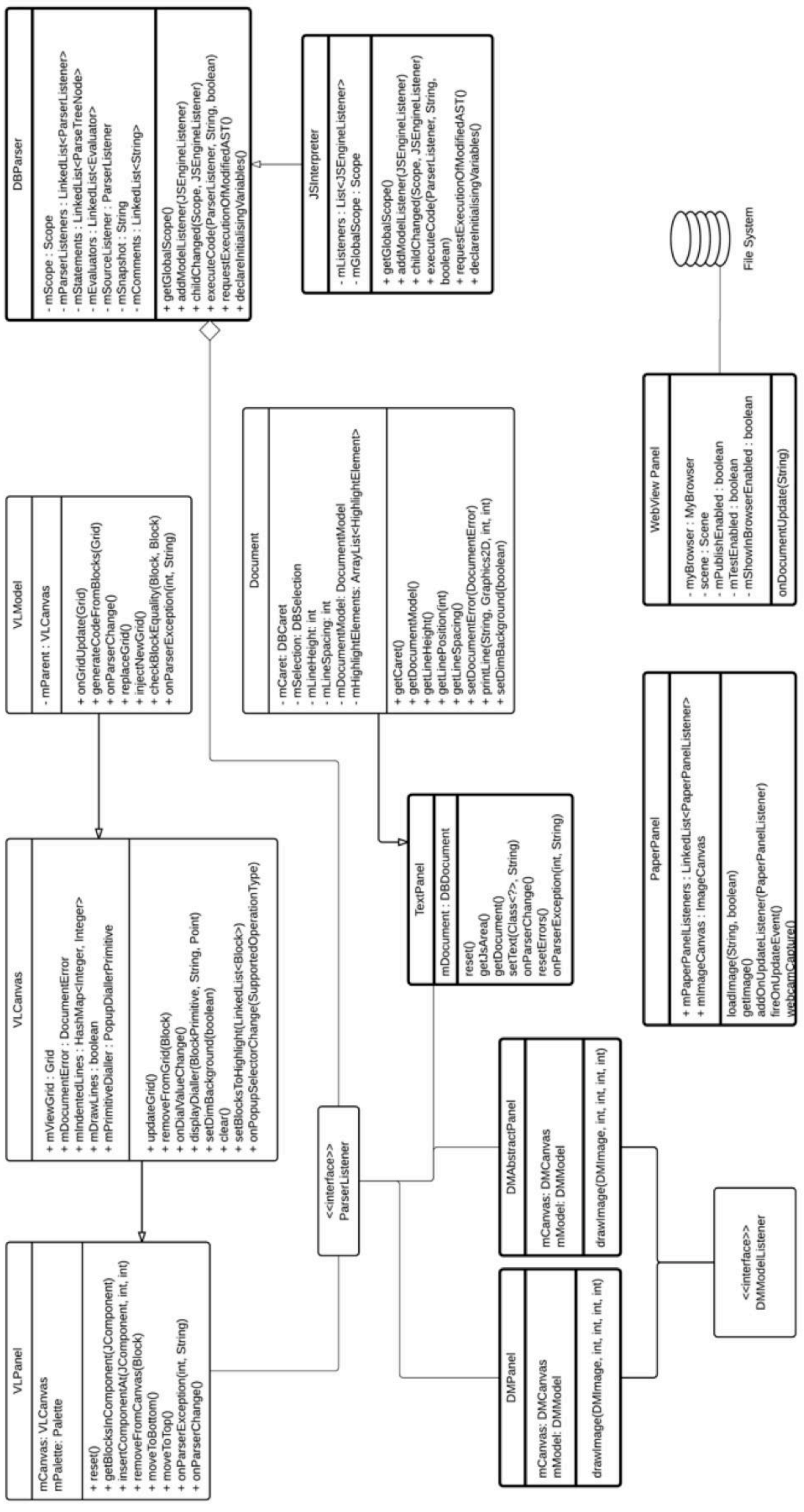

DBParser
- mScope : Scope
- mParserListeners : LinkedList<ParserListener>
- mStatements : LinkedList<ParseTreeNode>
- mEvaluators : LinkedList<Evaluator>
- mSourceListener : ParserListener
- mSnapshot : String
- mComments : LinkedList<String>
+ getGlobalScope()
+ addModelListener(JSEngineListener)
+ childChanged(Scope, JSEngineListener)
+ executeCode(ParserListener, String, boolean)
+ requestExecutionOfModifiedAST()
+ declareInitialisingVariables()
JSInterpreter
- mListeners : List<JSEngineListener>
- mGlobalScope : Scope
+ getGlobalScope()
+ addModelListener(JSEngineListener)
+ childChanged(Scope, JSEngineListener)
+ executeCode(ParserListener, String, boolean)
+ requestExecutionOfModifiedAST()
+ declareInitialisingVariables()
File System
VLModel
- mParent : VLCanvas
+ onGridUpdate(Grid)
+ generateCodeFromBlocks(Grid)
+ onParserChange()
+ replaceGrid()
+ injectNewGrid()
+ checkBlockEquality(Block, Block)
+ onParserException(int, String)
Document
- mCaret: DBCaret
- mSelection: DBSelection
- mLineHeight: int
- mLineSpacing: int
- mDocumentModel: DocumentModel
- mHighlightElements: ArrayList<HighlightElement>
+ getCaret()
+ getDocumentModel()
+ getLineHeight()
+ getLinePosition(int)
+ getLineSpacing()
+ setDocumentError(DocumentError)
+ printLine(String, Graphics2D, int, int)
+ setDimBackground(boolean)
WebView Panel
- myBrowser : MyBrowser
- scene : Scene
- mPublishEnabled : boolean
- mTestEnabled : boolean
- mShowInBrowserEnabled : boolean
onDocumentUpdate(String)
VLCanvas
+ mViewGrid : Grid
+ mDocumentError : DocumentError
+ mIndentedLines : HashMap<Integer, Integer>
+ mDrawLines : boolean
+ mPrimitiveDialler : PopupDiallerPrimitive
+ updateGrid()
+ removeFromGrid(Block)
+ onDialValueChange()
+ displayDialler(BlockPrimitive, String, Point)
+ setDimBackground(boolean)
+ clear()
+ setBlocksToHighlight(LinkedList<Block>)
+ onPopupSelectorChange(SupportedOperationType)
TextPanel
mDocument : DBDocument
reset()
getJsArea()
getDocument()
setText(Class<?>, String)
onParserChange()
resetErrors()
onParserException(int, String)
PaperPanel
+ mPaperPanelListeners : LinkedList<PaperPanelListener>
+ mImageCanvas : ImageCanvas
loadImage(String, boolean)
getImage()
addOnUpdateListener(PaperPanelListener)
fireOnUpdateEvent()
webcamCapture()
<<interface>>
ParserListener
DMAbstractPanel
mCanvas: DMCanvas
mModel: DMModel
drawImage(DMImage, int, int, int, int)
VLPanel
mCanvas: VLCanvas
mPalette: Palette
+ reset()
+ getBlocksInComponent(JComponent)
+ insertComponentAt(JComponent, int, int)
+ removeFromCanvas(Block)
+ moveToBottom()
+ moveToTop()
+ onParserException(int, String)
+ onParserChange()
DMPanel
mCanvas: DMCanvas
mModel: DMModel
drawImage(DMImage, int, int, int, int)
<<interface>>
DMModelListener

Figure 5.10: DrawBridge Architecture

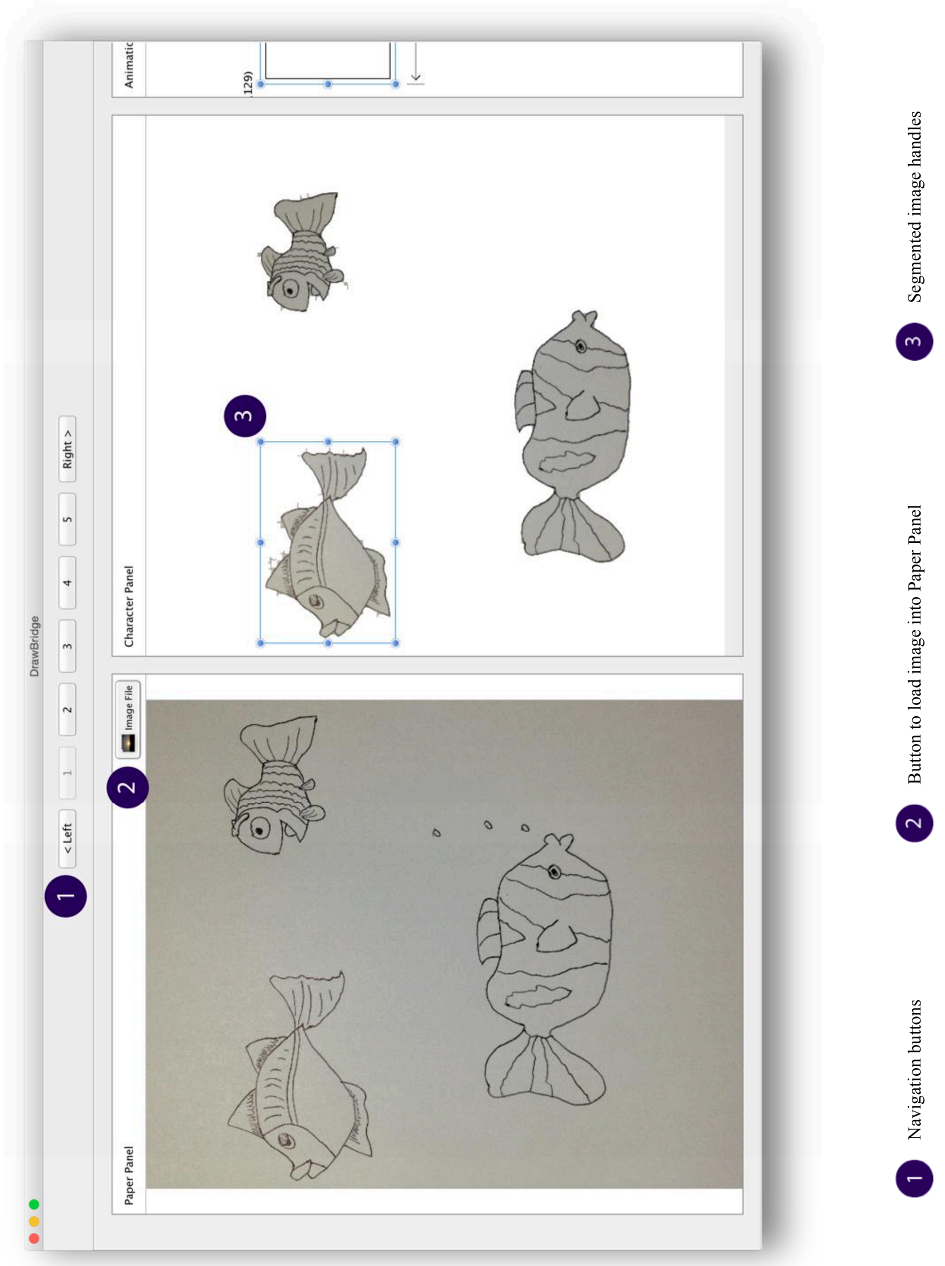


Figure 5.11: DrawBridge: Paper and Direct Manipulation

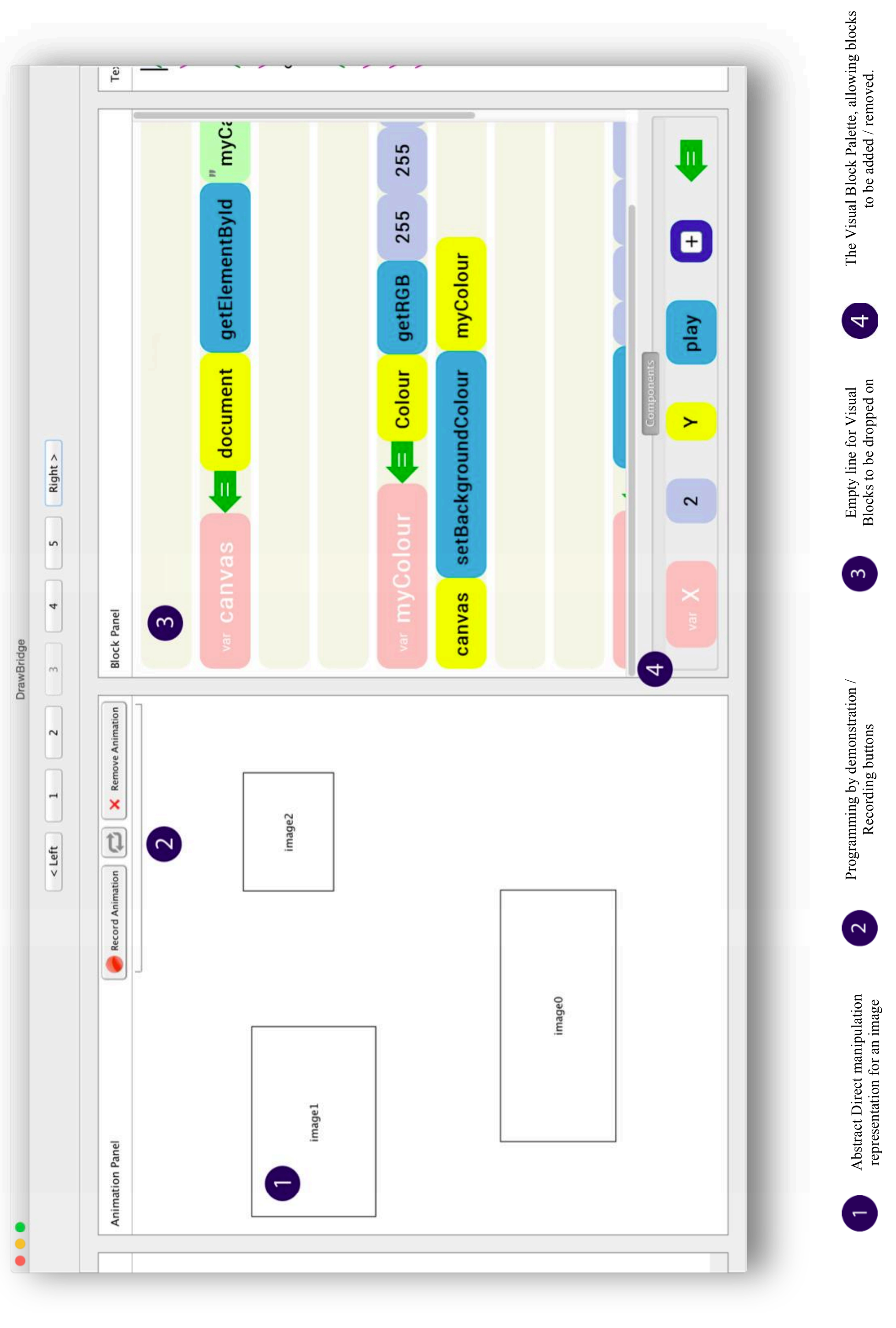
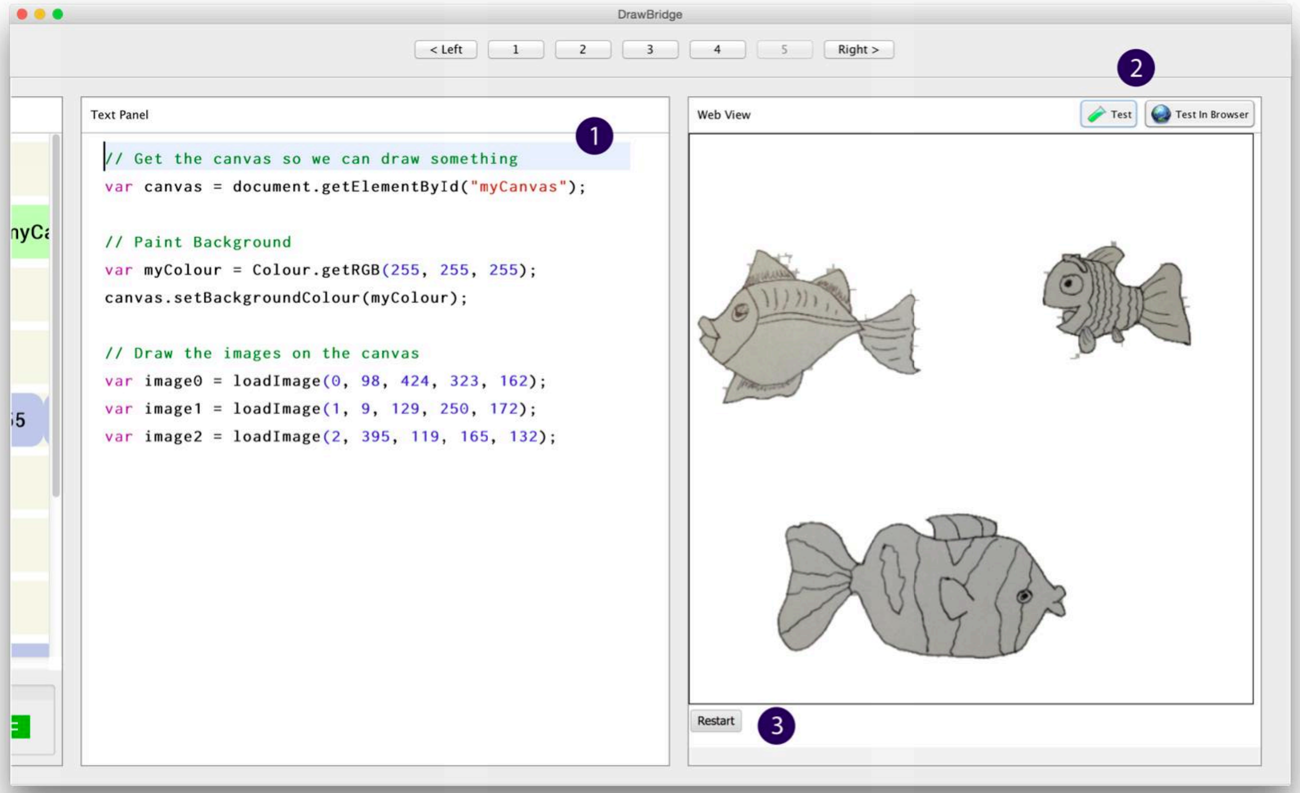


Figure 5.12: DrawBridge: Abstract Direct Manipulation and Visual Blocks



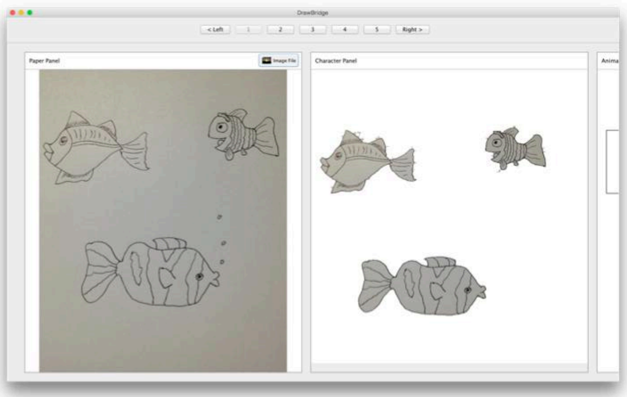
```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 98, 424, 323, 162);
var image1 = loadImage(1, 9, 129, 250, 172);
var image2 = loadImage(2, 395, 119, 165, 132);
```

1 Line highlighting to aid correspondence between Text and Visual Blocks

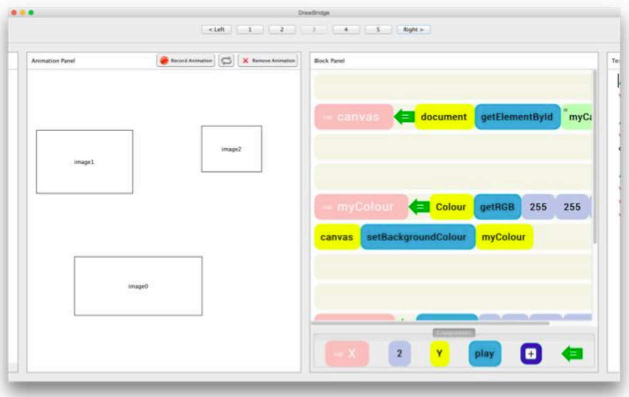
2 Testing buttons to allow users to check their work in an embedded web browser

3 Restart button allowing users to replay their code in the embedded web view

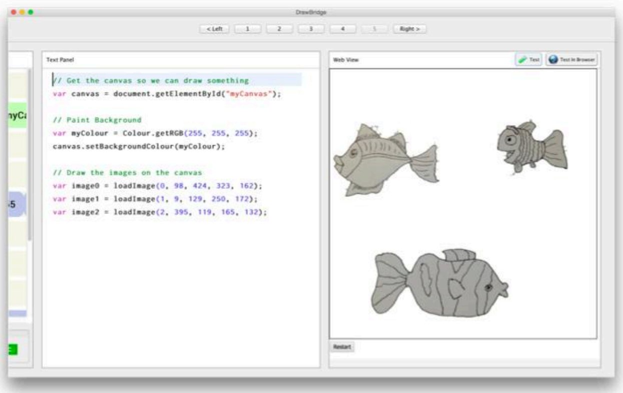
Figure 5.13: DrawBridge: Text and Web View



(a) First panel pair; **Left**: Paper Panel showing input image **Right**: Character Panel showing segmented images



(b) Second panel pair: **Left**: Abstract Character Panel allowing positioning of images and animation recording. **Right**: Visual Block panel allowing easy dial-based editing of parameters.



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 98, 424, 323, 162);
var image1 = loadImage(1, 9, 129, 250, 172);
var image2 = loadImage(2, 395, 119, 165, 132);
```

(c) Third panel pair; **Left**: Text Panel allowing users to edit DrawBridge code. **Right**: Web View Panel allow a web preview of animation.

Figure 5.14: Panel Pairs in DrawBridge

5.4.2 Paper Capture

To capture characters drawn on paper in a robust and repeatable fashion, DrawBridge uses computer vision techniques implemented using the OpenCV library. A reimplementation of Zhang's WhiteboardIt! algorithm for cropping and background removal of documents captured in an image (Zhengyou Zhang, 2002) was created to allow captured photos to be cropped and segmented in DrawBridge. However, Zhang's algorithm for background removal was replaced with the adaptive thresholding algorithm (Bradley & Roth, 2007), which is more robust for images with poor lighting conditions. The final steps for processing each input image are listed below:

1. Apply an adaptive threshold to create a new edge image.
2. Apply dilation and erosion to the edge image to join unconnected edges.
3. Run an active contour search over the edge image.
4. Filter identified contours by threshold area (0.5% of image area).
5. Use the resulting contours to segment the original image.
6. Create segmented images from each contour using pixel data from the original image.

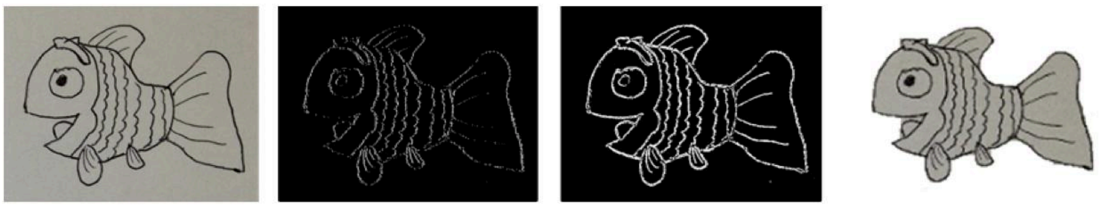


Figure 5.15: Example of Character Image Processing Pipeline

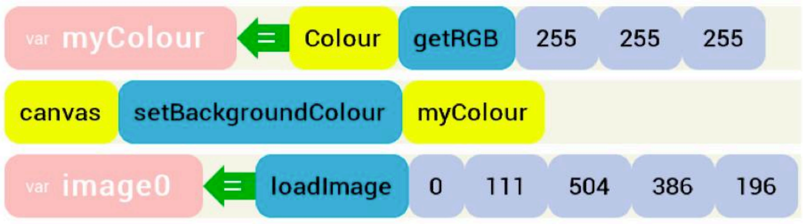
(a) Original image (b) Edge image after adaptive thresholding (c) Edge image after erosion and dilation (d) Character segmented from original image using identified contour.

This method of segmentation requires that users must follow certain guidelines when drawing characters so that they are correctly identified, including (a) giving each character a complete contour, (b) keeping the character bounds within the supplied image (i.e. no hitting the edges so as to meet point a), (c) making characters larger than 0.5% of the image area, (d) making sure character contours do not touch.

The accuracy and robustness of the character segmentation algorithm was tested using 25 different images created using a variety of lighting conditions and various drawing styles. An example of the final image-processing pipeline is shown in Figure 5.15. To import and apply image segmentation in DrawBridge, users click the "Image File" button on the Paper panel (Figure 5.11-1).

5.4.3 Visual Language Blocks

The visual block language was intentionally designed to be similar to a text representation, while retaining the usability advantages of visual languages. The example code below shows that each block is positioned on a single line, moving from left to right. Blocks are shaped as rounded rectangles, to help the user distinguish between blocks placed next to each other that are the same colour. Each block can be dragged out of position, and dropped into a new position using a yellow target highlighter that appears during dragging. Users can add new blocks by dragging them onto the panel from a palette at the bottom of the screen (see Figure 5.12-4).

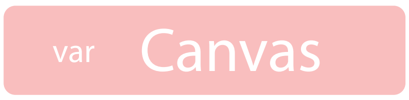
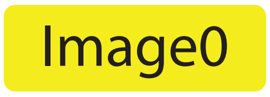
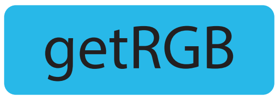
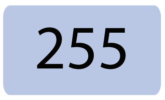


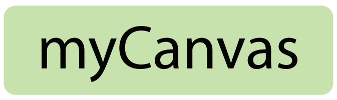
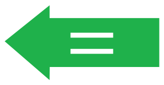
```
var myColour = Colour getRGB 255 255 255
canvas setBackgroundColour myColour
var image0 = loadImage 0 111 504 386 196
```

Figure 5.16: Example Visual Block Code

A list of visual blocks and their text equivalent is given in Table 5.4. Each block type has a corresponding language entity and a text equivalent. The colour of each block is matched as far as possible to the syntax highlighting.

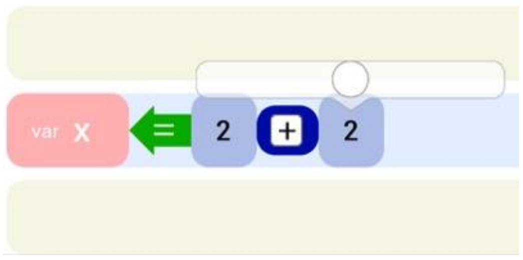
Table 5.4: DrawBridge Visual Blocks and Text Equivalents

Visual Block	Entity	Text Equivalent
 var Canvas	Declaration	`var canvas`
 Image0	Identifier	`image0`
 getRGB	Function / Method	`getRGB()`
 255	Primitive – Number	`255`

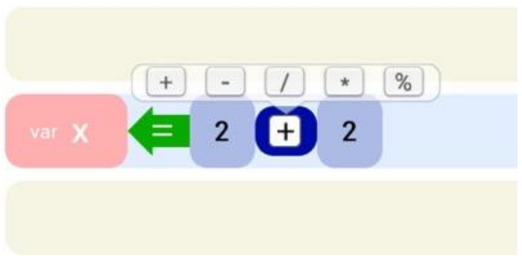
	Primitive - String	“myCanvas”
	Assignment	=
	Operator	+

Reducing the Viscosity of Visual Blocks

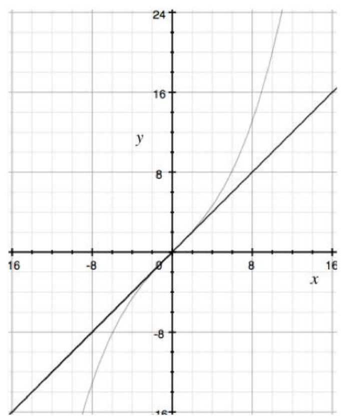
The primary editing mechanism for the visual block panel is to use the mouse to drag and drop blocks into position. Although the content of blocks can be edited using a keyboard, constant mode switching between mouse and keyboard may reduce students’ ability to tinker (Nash, 2012). To allow edits using the mouse, context-sensitive value-modifiers (Figure 5.17 a and b) were created to appear when the user hovers their mouse over a block. The number dialler allows users to modify any number by manipulating a transparent dial left to reduce the number and right to increase it. A function was applied to the dial distance to allow users to make large changes (Figure 5.17-c). Users are able to experiment with operators using the operator modifier.



(a) Number Dialler



(b) Operator Modifier

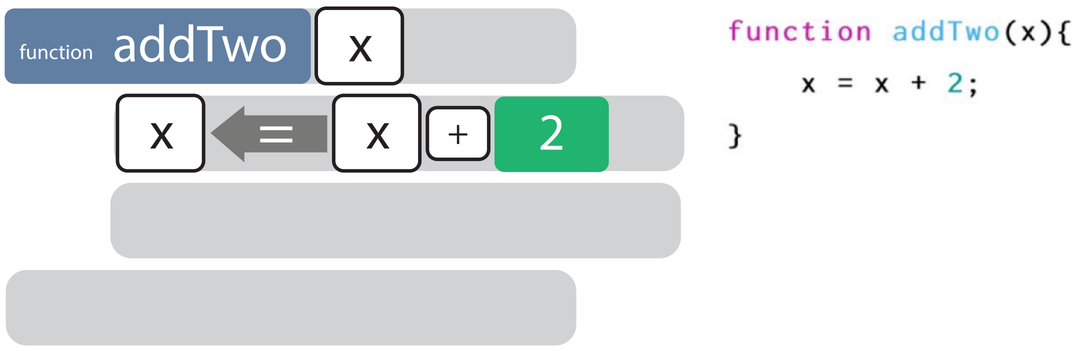


(c) Number Dialling Function $y = x + \frac{1}{100}x^3$

Figure 5.17: Transparent Context-sensitive Value Modifiers

Proximity Syntax

Some parts of the text-code had no obvious equivalent visual block representation. For example, curly braces are used in a JavaScript function to denote the start and end of functions, if loops and while loops. The lack of nesting in the visual block panel led to two options: (1) add a new block type to represent curly braces, or (2) use another visual cue to represent the content inside the braces. The former option does not make best use of the visual nature of the block representation, and is likely to increase error proneness in the representation. Therefore, to make it clear that blocks were inside a function or loop, they were indented. The usability implications of this are more complex than they might first appear. When creating a new function in the block language, the differentiation of what is inside the function, and what is outside is not obvious.



```
function addTwo(x){
    x = x + 2;
}
```

Figure 5.18: Example Function in Visual Block and Text Representation

An extra indented line is provided in the visual block panel to allow users to drag and drop new blocks to a new line inside the function.

5.4.4 Bi-Directional Support

The implementation for bi-directionality between representations was particularly complex due the requirement that all four interactive representations be up to date. In order to reduce latency during interaction, the existing Abstract Syntax Tree (AST) was regenerated as few times as possible. Figure 5.19 shows the execution architecture for the system. Updates made to the Direct Manipulation panels, and value modifiers on the visual block panel, modify the AST directly, rather than requiring reparsing. As a result, each object in these representations is required to maintain a reference to the AST, which can be used to propagate new modifications when the object is changed. The system must also differentiate between human interaction, and program execution, which moves the objects in the Direct Manipulation panel, to avoid infinite loops.

When the text panel is edited, or the structure of the visual blocks modified, the code is parsed to create a new AST. Each object on the Direct Manipulation panels is then updated with references to the new AST.

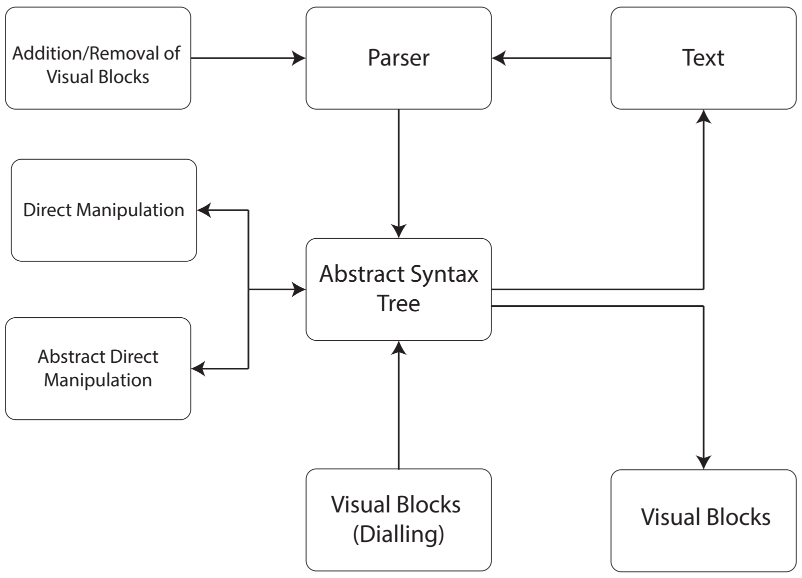


Figure 5.19: Simplified Bi-directional System Architecture

Synchronised Scrolling

When the number of lines in the program increases, the user is required to scroll to view the piece of the program they would like to edit. However, correspondence highlighting does not work if the corresponding part of the second representation is not visible too, which is often the case when two representations have different line heights. DrawBridge therefore employs synchronised scrolling between the visual block panel and the text panel.

5.4.5 Practical Liveness

A practical issue when using level-3 (edit-driven) and level-4 liveness is that when programs increase in size, the time taken to execute them can exceed the time between user modifications, forcing the user to wait until the previous execution has finished to see their latest modifications.

To avoid this problem, DrawBridge gives users the facility for edit-driven liveness *before* they create animations. This allows users to make live edits that can be viewed very quickly.

When users decide to add animation, edit-driven updates are turned off by default (but can be switched back on if required).

5.4.6 System Limitations

As mentioned previously, each modification of the visual block structure results in a regeneration of the AST. The parser used to generate the AST filters whitespace during the lexing process, and thus loses information about indentation, and whitespace. Therefore the updated text generated from the AST is a "clean" version of the code, which may be different from the users' previous text.

The animation playback in the Abstract DM Panel is implemented by modifying custom JComponent objects rendering each image using Graphics2D in a JPanel. The frame rate of animation is limited by the power of the computer's processor. To ensure time-accurate playback, the system records the time required to render each frame, and generates new frames based on the expected render time.

5.5 Chapter Summary

This chapter presented central design decisions made during the development of a prototype MER system, DrawBridge, which was created to facilitate the study of students' acquisition of notational expertise in MER systems.

The design of the system followed heuristics created using the MoRA framework presented in Chapter 3 to allow students to make the transition between five major representations: paper, Direct Manipulation, Abstract Direct Manipulation, visual blocks and text. The design of each representation, together with examples of alternative designs, was discussed in detail. In particular, a mechanism for using Direct Manipulation to create programming code that can scaffold the users progress via Programming by Demonstration was presented. Further, the design of a novel visual language was presented, which is intentionally similar to text notation to allow students to more easily make the transition to text-based representations.

Representations are displayed on the screen in pairs, allowing students to view correspondences between representations that were deliberately designed for each pair. The usability of each pair of representations was analysed using the Cognitive Dimensions (CDs) of Notations framework to ensure that, when used as a pair to carry out incrementation and exploratory design tasks, the usability of the combined representations was maximised.

Further, the motivation of the system was described, together with desirable execution properties intended to encourage experimentation and progressive evaluation.

The second half of the chapter discussed the practical implementation of DrawBridge, which was written in Java. The system uses a custom interpreter to execute the updated or regenerated underlying abstract syntax tree, in order that all interactive representations are kept up to date. This allows students to make edits in familiar representations when unsure how to edit a more complex representation. Finally, the process by which robust computer vision techniques are used to segment characters drawn on paper is presented, followed by the implementation of support features, such as dialling, which further reduce the viscosity of the visual blocks interface, and encourage quick experimentation by allowing users to avoid switching between input devices.

Chapter 6 Manipulation of MERs to Encourage Notational Expertise

6.1 Introduction

The previous chapter described the design and implementation of DrawBridge, a novel educational programming environment that uses Multiple External Representations (MERs) to allow students to make the transition from accessible, low-abstraction tangible representations, to highly abstract, symbolic programming representations.

This chapter presents two studies that seek to explore two major design decisions made when creating DrawBridge that both concern the pairings of representations and their subsequent correspondences. In particular, the studies examine groups using alternate orders of symbolic representations, and groups with and without low-abstraction representations in DrawBridge to investigate whether the representation-transition strategy affects acquisition of Notational Expertise (NE).

In order to investigate and evaluate the design goals of DrawBridge, each study was conducted in an authentic context in which problems and interactions that arose during the sessions could be recorded and discussed in addition to results. In the case of DrawBridge, the context is a typical classroom environment with existing classes of students and existing computing lesson formats. A study in this context would require an experimental design that takes real world variation into account – a quasi-experiment.

The main benefit of a quasi-experimental design is that it enables the study of groups where randomised assignment to treatment and comparison groups has not been used (Campbell & Stanley, 1967). Each study presented in this chapter enables the exploration of questions relating to design decisions made in DrawBridge and the way in which students use the system. Each study contributes a discussion of problems and interactions that arose during the sessions, how they were addressed, and the way in which they affected outcomes. Investigation and discussion regarding the results of these studies will be used to further refine the design of DrawBridge and provide both guidance for both researchers carrying out interventions using MER systems and designers of new MER systems.

The structure of this Chapter is as follows. Section 6.3 describes a pilot study in which the procedure, study materials and target user interaction are refined. Section 6.4 presents the proposed design of the study, including descriptions of each lesson conducted in the study. Section 6.5 presents the results of each study, and Section 6.6 discusses all three sets of results in the context of the research questions presented in Section 6.4. Section 6.7 presents a follow up think-aloud study with three participants. Finally, conclusions and future work are presented in section 6.9.

6.2 Alternative Transition Strategies in DrawBridge

The MoRA framework, presented in Chapter 3, assisted the creation of candidate transition strategies to allow students to make the transition between low-abstraction tangible representations to high-abstraction symbolic representations. The resulting strategies were supported by interviews carried out with teachers regarding current classroom practice, analysis of representation pairs using the Cognitive Dimensions (CDs) of notations framework, and studies presented in Computer Science Education literature that describe the use of novice programming tools to scaffold understanding of text representations (e.g. Mishra, 2014). As a result of this analysis, DrawBridge was developed to provide students with a sequential transition between five core representations: Paper, Direct Manipulation, Abstract Direct Manipulation, Visual Blocks, and Text. However, alternative combinations and configurations of representations exist that merit investigation.

6.2.1 The Requirement for Low-Abstraction Representations

The design rationale for the inclusion of the first three representations in DrawBridge was to act as a strong motivation to engage with the system and to scaffold student knowledge by increasing student familiarity with highly abstract representations over time. However, the extent to which the design has been successful is not yet known. It is possible that using animation of hand-drawn characters may not provide sufficient motivation to engage students in programming activities. It is also possible that Paper and Direct Manipulation representations may be unsuccessful in helping to increase students' familiarity with abstract representations, and these representations may obstruct or distract students, resulting in difficult transitions to later representations. These possibilities raise the research question of the extent to which low-abstraction representations improve students' acquisition of Notational Expertise in MERs. This question will be addressed later in the chapter.

6.2.2 The Order of Symbolic Representations

As described in Chapter 5, DrawBridge presents representations sequentially from left to right (minimal-abstraction to unlimited-abstraction), and displays two representations on the screen at once. Students can move through representations using navigational arrows, but the order in which they encounter representations is fixed (Table 6.1).

Table 6.1: Representation Pairs in Visual-First

Pair	Left	Right	Correspondence
1	Paper	DM	The position of DM objects is initialised at the position in the image
2	DM	Abstract DM	Location, size and selection of objects is mirrored.
3	Abstract DM	Visual	Modified properties are highlighted in both representations.
4	Visual	Text	Modified properties are highlighted in both representations.
5	Text	Final Output	-

(DM = Direct Manipulation)

Visual-First

Candidate transition strategies created using the MoRA framework resulted in the order of representations shown in Table 6.1 and Figure 6.1. In this order, referred to as "Visual-First", or VF, students first encounter visual blocks, which have several usability advantages when compared to text representations, such as *low viscosity* $^{(CD)}$, and high *role-expressiveness* $^{(CD)}$ (see Chapter 5). This order mirrors a typical classroom transition, identified during teaching interviews (see Chapter 4), where visual languages are the preferred starting point for novice programmers.

In this order, users would make the transition to the abstract DM and visual block representation pair, viewing correspondences between the objects on the left and visual blocks on the right (as in Table 6.1 pair 3). The next transition would move the user to the visual block and text representation pair (pair 4). In this order, it is not possible to see correspondences directly between the abstract DM and text representations, due to the visual block representation between them.

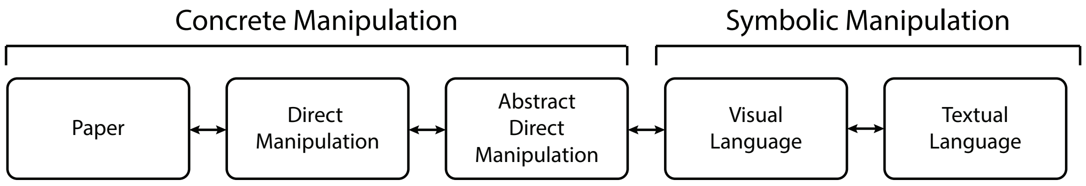


Figure 6.1: Visual-First DrawBridge Representation Order

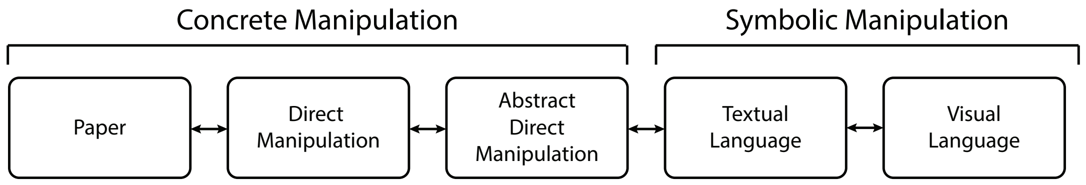


Figure 6.2: Text-First DrawBridge Representation Order

Text-First

An alternative order, shown in Figure 6.2 would keep the order of the first three representations, but swap the text and visual block representations. This text-first strategy, or TF, would allow students to modify properties of their characters in the abstract DM representation while viewing corresponding changes in the text code, allowing students to infer the structure and meaning of the code. In this order, the visual block representation could be used to support modification of text. This leads to two further research questions that will be investigated in this chapter. First, to what extent does the order of Visual and Text representations in DrawBridge affect students' acquisition of Notational Expertise; and second, to what extent does the combination of the use of low-abstraction representations, and the order of Visual and Text representations, change acquisition of Notational Expertise?

6.3 The Pilot Study

A pilot study was conducted to refine the design of the study, further test DrawBridge with users, and verify that DrawBridge functioned as expected when running on school infrastructure, which is typically hostile to new software (see interviews with teachers in Chapter 4 and Hepburn and Buley (Hepburn & Buley, 2006)).

The pilot study was carried out in an after-school IT club in Cambridgeshire over two sessions, each lasting an hour and a half. Club attendance was voluntary and no female students were present.

Pilot Study Procedure

Participants were given an introductory questionnaire (see Appendix B), which recorded demographics, and answers to the following questions:

- Have you made animations on the Computer before? If so, how?
- Have you programmed a computer before? If so, how?
- If you have programmed, how long have you been programming for?
- Have you heard of/used JavaScript?
- Do you enjoy Computing? What are the best and worst parts?
- Which of these languages have you used? Python, JavaScript, Java, C++, Visual Basic, Scratch, Alice, LISP, C#, C

On completion of the questionnaire, participants were given a pre-test, which assessed their understanding of JavaScript using multiple-choice questions (see Appendix B). After using DrawBridge for 40 minutes, students were given a post-test that contained the same questions, but in a different order, and with different variable names and number values. Ethical approval for the pilot study was obtained from the Computer Laboratory Ethics Committee.

Pilot Study Results

Three participants completed the study in full. A further three students, who were unable to use DrawBridge in the first visit due to school network limitations, completed the questionnaire and gave feedback on the pre-test. Participants were aged between 12 and 13, and had a range of programming experience (see Table 6.2).

Participant	Age	Programming Experience
P1	13	Scratch
P2	12	JavaScript, Python, HTML
P3	13	N/A
P4	12	Python
P5	12	N/A
P6	12	Scratch, Alice, Python, Java

Table 6.2: Pilot Study Participants

Participants completed the questionnaire without raising any concerns. However, when prompted to see if there were any difficult questions, two participants stated that they found the last question, regarding programming language experience, difficult due to not recognising many of the languages listed.

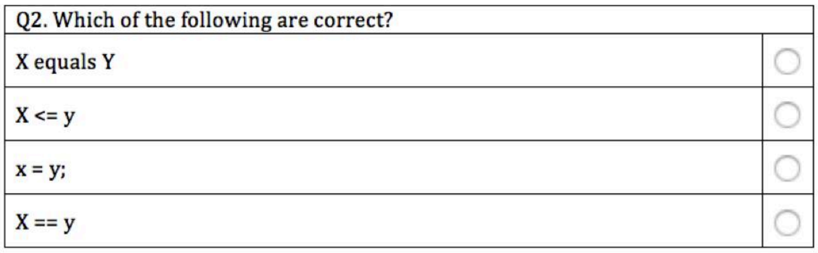
One participant said they were unsure about all of the questions in the pre and post-tests, suggesting low confidence or a fragile level of self-efficacy (as defined in Chapter 2). When encouraged to attempt the questions, the participant correctly answered the majority of questions. All three participants diligently followed the tasks and completed their animations within the allocated time.

Three technical problems were resolved during the session and recorded as bugs to be fixed before the main study.

Participant Interview

During the second pilot study, there was opportunity to carry out an interview with one participant (P2). During the interview, the participant described their experience with DrawBridge, the design of the questionnaire, and pre and post-test questions. The participant stated that drawing characters to be used in animations on paper was particularly motivating and unlike anything they had done before, suggesting that tangible and Direct Manipulation representations provide sufficient motivation for students to engage in programming tasks.

During reflection on their use of DrawBridge, the participant stated that they did not understand the meaning of the "Tween" function, which is an animation term that refers to individual frames of animation created in between two "key" frames. When asked what a better name might be, the participant agreed that "setTimeToDestination" would be easier to understand.



Q2. Which of the following are correct?	
X equals Y	○
X <= y	○
x = y;	○
X == y	○

Figure 6.3: Example of Pilot Study Assessment Question

Finally, the participant described his thought process while answering questions in the pre and post-test. The first question, shown in Figure 6.3, asked the participant which statement gave "x" the value of "y". The participant had answered "x <= y". When asked why, he explained that the visual block, which shows a right-to-left arrow, resembled a less than or equals sign. The arrow (shown in Figure 6.4) was designed to avoid the classic confusion that arises between the "=" in programming that represents *transfer*, and the "=" in mathematics that represents *equality* (McIver & Conway, 1996) (McIver, 2000) (Dehnadi & Bornat, 2006). Although this problem occurred, the appearance of the block was not changed, as the expected benefits of including the visual arrow were expected to outweigh the likelihood of confusion.

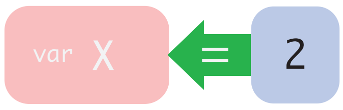
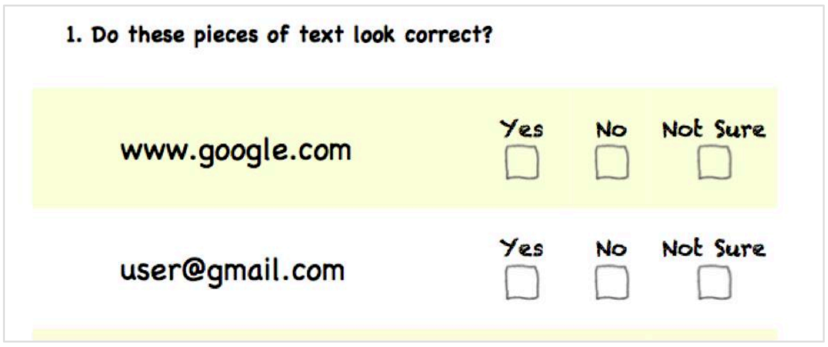


Figure 6.4: Visual Block Assignment Arrow

Changes to Procedure

Minor adjustments were made to the questionnaire, pre-test and post-test, to reduce potential intimidation. First, two "starter" multiple-choice questions regarding valid web addresses were added to the pre and post-test to support student confidence. The style of the assessments was also adjusted to be less formal by adding coloured cell backgrounds, using an informal font, removing table borders, and reducing the amount of text the student was required to read (see Figure 6.5).

The "tick if correct" format of the pre and post-tests allowed participants to simply avoid answering, which led to uncertainty as to whether they were not sure or thought it was incorrect. To address this, the question format was modified to a "Yes", "No" and "Not Sure", option-based style. This format encourages participants to provide an explicit answer for each question.



1. Do these pieces of text look correct?

www.google.com	Yes ☐	No ☐	Not Sure ☐
user@gmail.com	Yes ☐	No ☐	Not Sure ☐

Figure 6.5: Example of an Updated Assessment Question

The pre and post-test syntax test did not give insight into why the participant answered as they did. It also lacked any form of translation, which is a key part of the definition of Notational Expertise (NE). To fully assess participants' grasp of NE, translation question were added, which asked participants to translate visual-blocks into text, and text into visual-blocks.

During the pilot study, it was clear that the workflow of the study could be improved. For example, participants answered pre-test questions before completing their drawings, which had to be captured using a camera and transferred to the computers, taking several minutes. To reduce wasted time, the study procedure was modified so that participants completed their drawings before the pre-test so that drawings could be captured and processed while they completed the pre-test.

In first pilot study session, it was found that DrawBridge could not write temporary files to the directory it was running from due to restrictions enforced by school IT infrastructure. This problem blocked DrawBridge from storing copies of images, logs and webpages. DrawBridge was subsequently modified to write temporary files to users' local "My Documents" directory.

6.4 Quasi-experiment Design

A quasi-experiment was designed to further explore the research questions described earlier in this chapter. The between-subjects study measured the effect of two factors: (a) the inclusion of concrete representations, and (b) the order of representations in DrawBridge.

For brevity, the first three DrawBridge representations: Paper, Direct Manipulation and Abstract Direct Manipulation, will hereafter be referred to as the "concrete" representations.

The study investigated the following questions: (1) To what extent will the acquisition of Notational Expertise in participants who use versions of DrawBridge with concrete representations be more than participants who use versions of DrawBridge without concrete representations; (2) To what extent will the acquisition of Notational Expertise in participants using Visual-First versions of DrawBridge be more than participants using Text-First versions of DrawBridge; and (3) To what extent will there be an interaction between the inclusion of concrete representations and the order of symbolic representations.

6.4.1 Participant Selection

A complete class of twenty-one school students were recruited from an independent school for girls in Cambridgeshire. Participants were studying in Year 7, aged between 11 and 12, and had a range of academic abilities.

C-VF	**C-TF**
Includes concrete representations Visual block syntax first	Includes concrete representations Text syntax first
VF	**TF**
Excludes concrete representations Visual block syntax first	Excludes concrete representations Text syntax first

Figure 6.6: Lesson 1 Participant Grouping

The quasi-experiment was conducted during two lessons with the same class over a two-week period. In Lesson 1 participants were divided into four groups (see Figure 6.6). In Lesson 2, all participants were given the same version of DrawBridge, which contained all representations, with a Visual-First order (C-VF). Three participants were absent, and the lesson was conducted with eighteen participants.

Groups were allocated according to seating in Lesson 1 before students entered the classroom. Each group was placed together to reduce possible demotivation of participants who were not able to take part in drawing and animation activities.

6.4.2 Procedure

In Lesson 1, participants were given a pre and post-test assessment to measure Notational Expertise (NE). Each assessment contained two questions: A1 asked participants to evaluate a the validity of a JavaScript statement by responding with "Yes", "No" or "Not sure"; A2 asked participants to translate from visual-blocks to text code, and from text code to visual blocks (see Appendix B).

After the pre-test, each group was given a task list that could be used with their specific version of DrawBridge. As VF and TF only had a version of DrawBridge with two representations, they could not use characters drawn on paper to create animations. Both groups were therefore given two translation tasks to complete, which consisted of the following steps:

1. Transcribe the given program to the left representation
2. Read the generated representation on the right
3. Delete the content from both representations

4. Attempt to re-create the representation on the right
5. Compare the resulting representation on the left with the program given.

Groups C-TF and C-VF had versions of DrawBridge that contained all representations, including the first three concrete representations (Paper, Direct Manipulation and Abstract DM) and symbolic representations (Visual and Text). They were given the following tasks:

1. Draw and load an image into DrawBridge.
2. Use Direct Manipulation to modify the location or size of a resultant character.
3. Use the Abstract DM representation to create a simple animation.
4. Use the next representation to add an extra step to the animation.
5. Use the last representation to modify and extend animation.

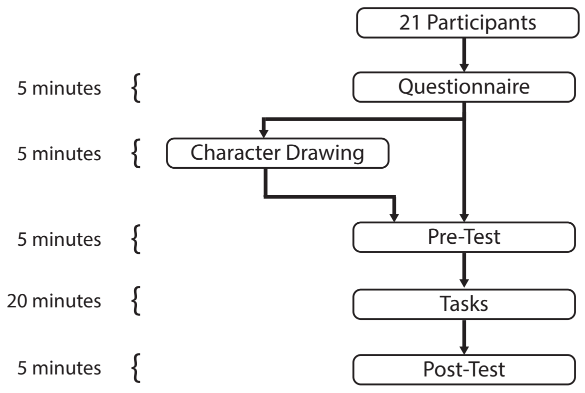


Figure 6.7: Lesson 1 Workflow

Lesson 2

In Lesson 2, all participants used the C-VF version of DrawBridge. The procedure for this lesson was similar to Lesson 1, except that, due to time constraints, there was no pre-test. Each participant was given a post-test, which was similar to the tests in Lesson 1, but had modified question order, variable names and number values. After completing the post-test, participants were given a post-study questionnaire asking them about their motivation, and experience with DrawBridge (see Figure 6.8 and Appendix B).

Preliminary results from Lesson 1 found that there were extremely low levels of participation in A2 of the assessment. In an effort to increase participation in Lesson 2, participants were encouraged to try to attempt the translation question, even if they were not sure.

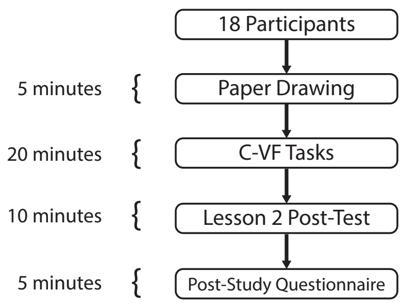


Figure 6.8: Lesson 2 Workflow

6.4.3 Data Collection

The data collected included results from the background questionnaire, pre and post-test data, student drawings, task notes, motivation questionnaire, and post-study questionnaire data.

Questionnaire

Participants were given an introductory questionnaire that collected details on gender, age, computing skill level, computing enjoyment level, and experience with animations, building webpages and programming.

Pre-test and Post-Test

Participants were given a pre-test containing two assessments: A1, which was a syntax assessment that consisted of 10 questions related to text syntax; and A2, which was a translation assessment that consisted of 8 translation questions. Of the 8 parts of A2, 4 were text to visual translations and 4 were visual to text translations. Each part addresses different parts of the syntax (see Table 6.3 and Appendix B).

Translation Example	Syntactic Features								
	Brackets	Parameters	Semi-Colon	Equals	Operators	Whitespace	Commas	Dot Syntax	String Quotes
`setTimeToDestination(500);`	☑	☑	☑						
`var x = 2 / 2;`			☑	☑	☑	☑			
`loadImage(2, 541, 173, 238, 168);`	☑	☑	☑			☑	☑		
`document.getElementById("myCanvas");`	☑	☑	☑					☑	☑

Table 6.3: List of Syntax Features

Participants were given a post-test after completing their tasks. The post-test was identical to the pre-test, but questions were given in a randomised order, and values were modified in an effort to reduce learning effects.

Instrumentation

DrawBridge was instrumented to capture user activity during the study. The activity captured included:

1. Length of time each pair of representations was visible on screen.
2. When DrawBridge was minimised and restored.
3. Syntax Errors (e.g. missing a bracket)
4. Lint Errors (e.g. using an undeclared variable)
5. User interface features (dialling)
6. Animation state changes
7. Animation Recording
8. Button clicks

Each log was recorded with a timestamp. Instrumentation data was captured using both an analytics service and local logs in an effort to avoid any loss of data due to issues with the network connection or local file access. Users' segmented images, and final animations were stored locally, and therefore could be captured at the end of the study.

Participants were observed during completion of assessments and tasks in the study. Students getting stuck or asking questions were assisted and recorded to assist analysis of results.

6.5 Results

6.5.1 Lesson 1

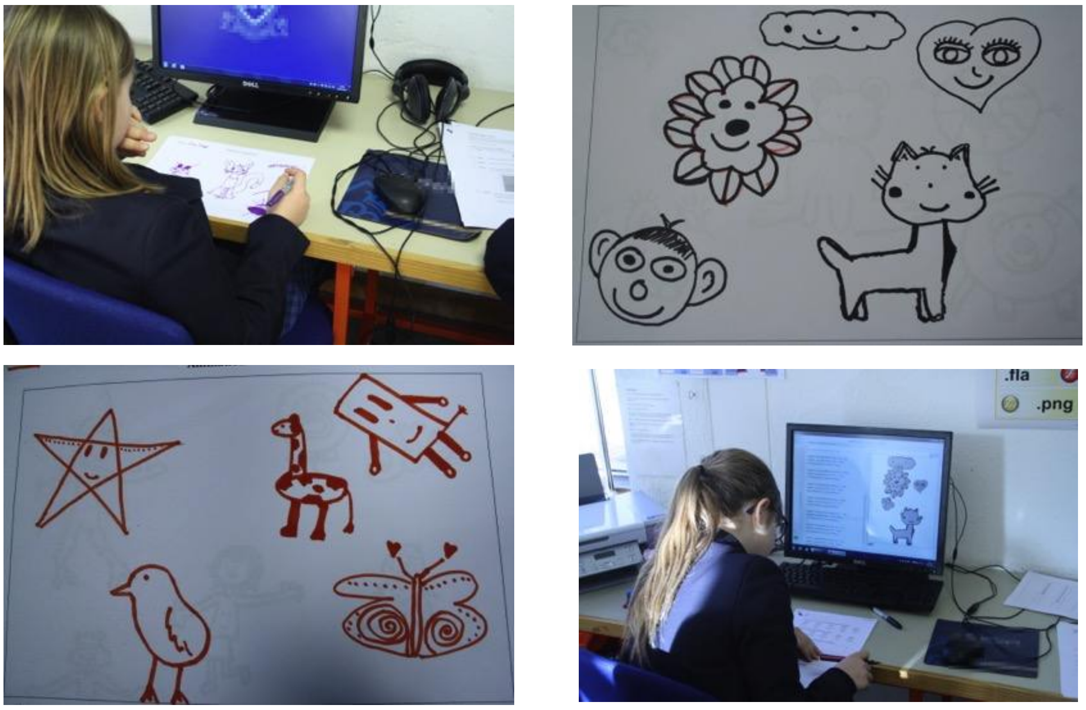


Figure 6.9: Example of Participant Characters on Paper

Participant Background

Only half of participants reported previous programming experience in the questionnaire despite the teacher reporting that they had all used Light-Bot, a novice programming environment in which users control a robot to complete specific tasks by arranging symbolic instruction blocks in the appropriate order. Some participants were unsure as to whether they had programmed before due to a lack of a clear definition and stated that they had no programming experience, despite also stating that they had "animated" things in Scratch. Participants reported a normal distribution of computing skill, with 2 students reporting poor skill and 2 students reporting excellent skill (see Figure **6.11**). All participants reported positive or indifferent enjoyment of computing (see Figure **6.12**).

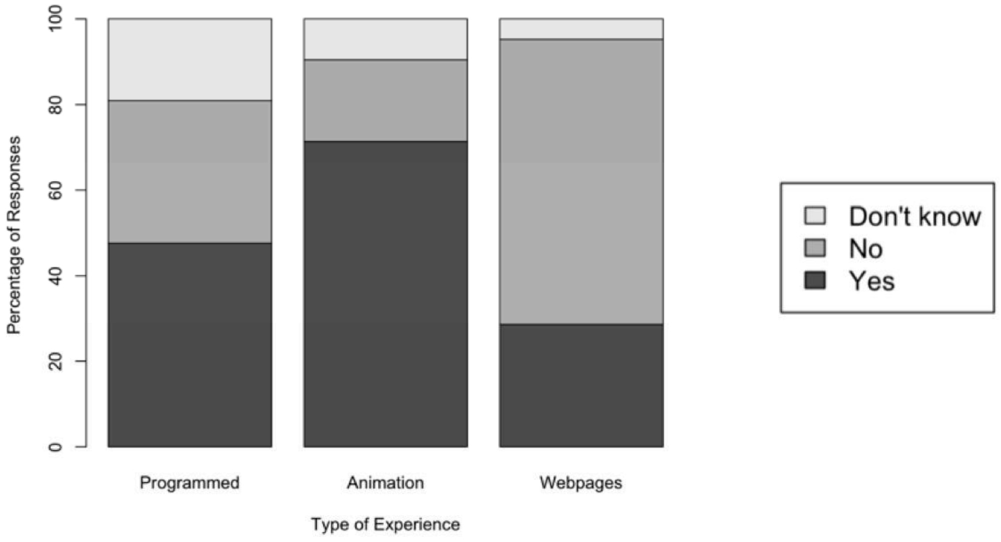
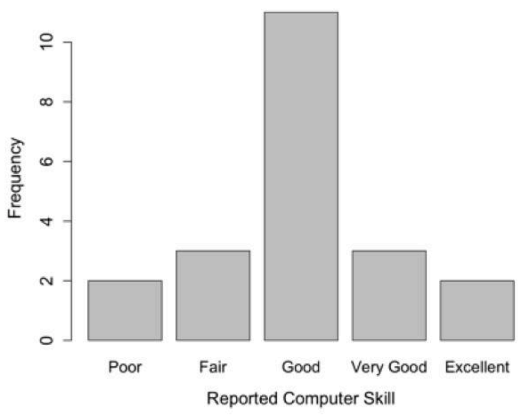


Figure 6.10: Reported Experience



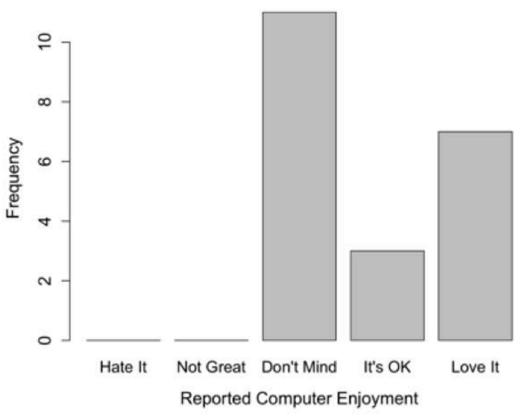


Figure 6.11: Reported Computer Skill

Figure 6.12: Reported Computer Enjoyment

Four of the six participants who reported they had created a webpage before could not remember what tools they had used to create the page. It is therefore likely that their experience was not substantial enough to be influential in this study.

The pre-test and post-tests consisted of two parts:

A1. Text syntax assessment

A2. Translation assessment

A1: Text Syntax Assessment

All participants answered A1 in both the pre-test and post-test. The two "starter" questions were answered correctly 90% of the time in the pre-test, and 70% in the post-test, despite being identical in syntax. Participants who classified themselves with "Very Good" or "Excellent" skill in the pre-questionnaire scored 5.66 and 6 on average, with participants responding "Good" and "Fair" scoring 3.64 and 3.67, and participants responding "Poor" scoring 1.5. However, a Kruskal-Wallis test found no significant difference between pre-test results and participants' reported computing skill ($p = 0.34($.

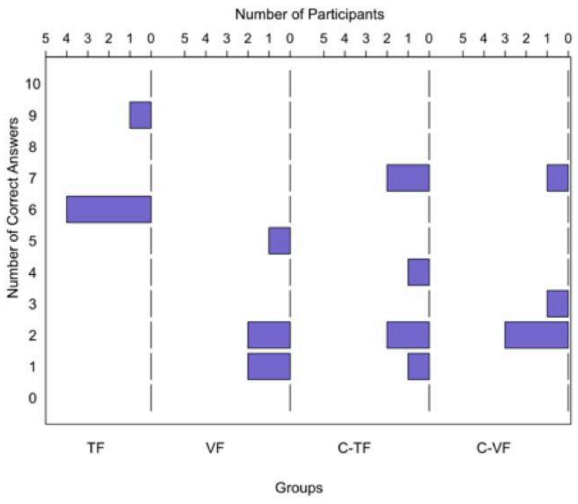


Figure 6.13: Pre-Test Correct in Syntax Assessment

In A1 of the pre-test group TF achieved the highest mean score of 6.6 when compared to the other groups VF, C-TF and C-VF, who scored means of 2.2, 3.2 and 3.8 respectively. An ANOVA showed there was a significant difference between groups $F(3, 17) = 4.22$, $p = 0.021$, which appears to be due to the high performance of TF (see Figure 6.13).

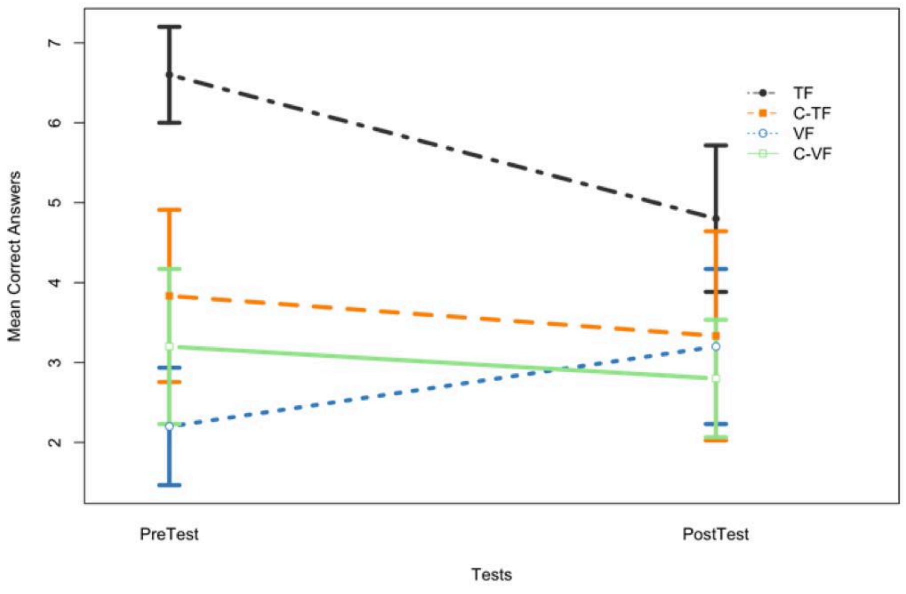


Figure 6.14: Changes in Pre-Post Syntax Assessment

The largest change between pre-test and post-test can be seen in Group TF, whose score decreased by 1.8. This change is possibly due to a regression towards the mean for Group TF, and is likely to have contributed towards an overall decrease in Text-First groups on average. The mean score of Visual-First groups increased, apparently due to improvement in Group VF. An ANOVA with post-hoc TukeyHSD found that there was no significant difference in visual-first groups between the pre and post-test: ($p = 0.99$) (see Figure 6.14 and Figure 6.16).

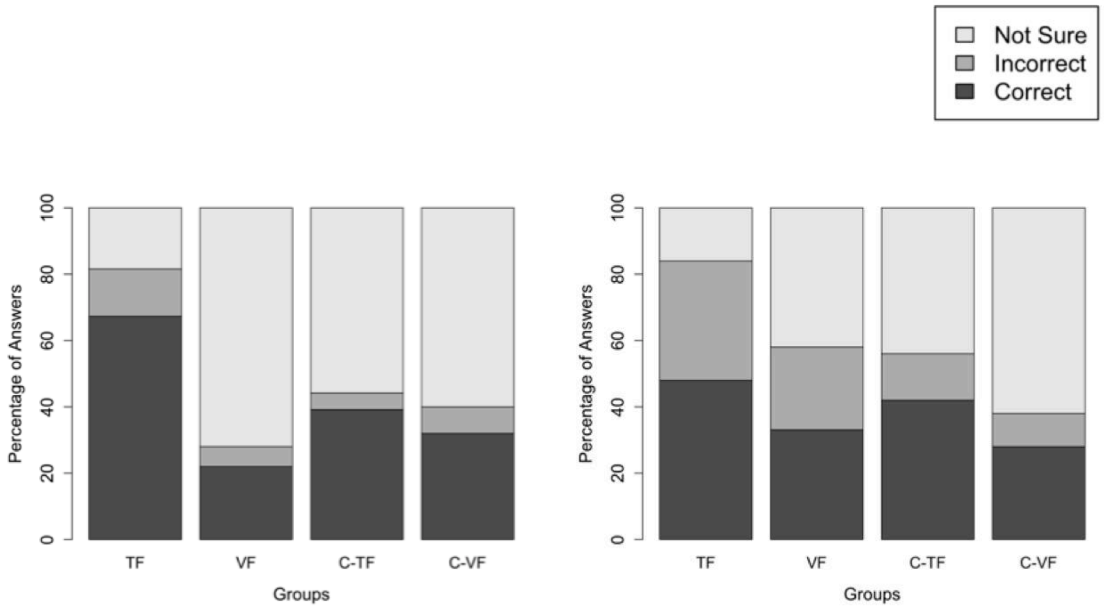


Figure 6.15: Lesson 1 Answer Categorisation by Group
Left: Pre-Test **Right**: Post-Test

Despite non-concrete groups outperforming concrete groups in both the pre-test and post-test on average (see Table 6.4), a one-way ANOVA showed there was no significant difference in improvement between groups with and without concrete representations: $F(1,17) = 0.002, p = 0.962$. The difference in means appears to be primarily due to the strong results of Group TF in both the pre-test and post-test.

Table 6.4: Mean Syntax Assessment Results

(± Confidence interval at 95%)

Group	Pre-Test	Post-Test	Change
TF	6.6 ± 1.18	4.8 ± 1.8	-1.8 ± 2.96
VF	2.2 ± 1.44	3.2 ± 1.9	1 ± 1.04
C-TF	3.83 ± 2.11	3.33 ± 2.56	-0.5 ± 3.74
C-VF	3.2 ± 1.9	2.8 ± 1.44	-0.4 ± 1.42
Non-Concrete	4.4 ± 1.943	4 ± 1.54	-0.4 ± 1.94
Concrete	3.5 ± 1.57	3.09 ± 1.69	-0.45 ± 1.76
Visual-First	2.7 ± 1.35	3 ± 1.31	0.3 ± 1.39
Text-First	5.09 ± 1.69	4 ± 1.83	-1.09 ± 2.02

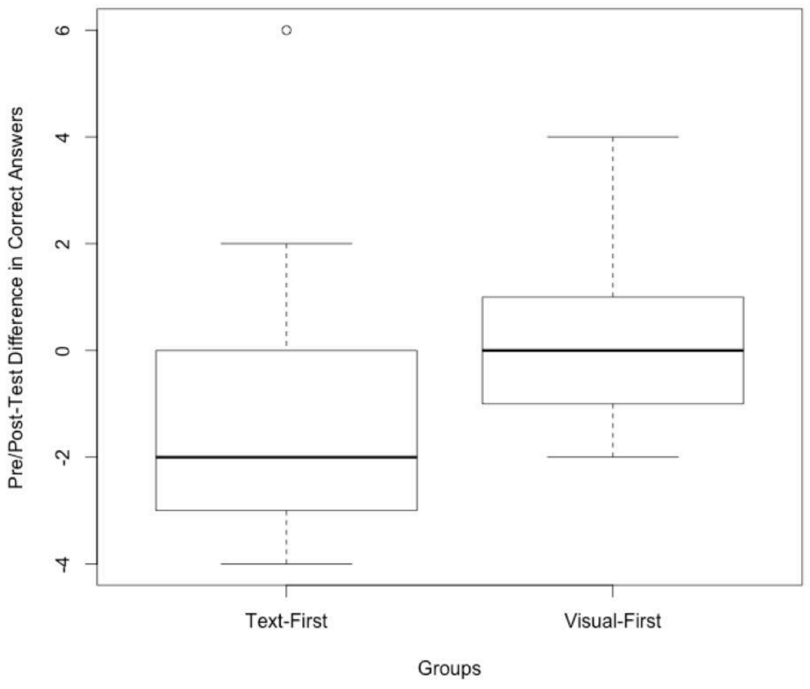


Figure 6.16: Change in Text-First and Visual-First Groups between L1 Pre-Test and Post-Test

Normalised results shown in Figure 6.16 and Figure 6.17 emphasise the change scores between pre-test and post-test, thereby addressing threats to internal validity caused by the high results achieved by Group TF in the pre-test, in accordance with non-equivalent group analysis (May, 2012). The biggest changes appear in Group TF, which diminish by approximately 2 marks on average, and VF, which improve by 1 mark on average. Post-hoc TukeyHSD tests showed that these changes were non-significant TF: ($p = 0.90$). VF:($p = 0.99$).

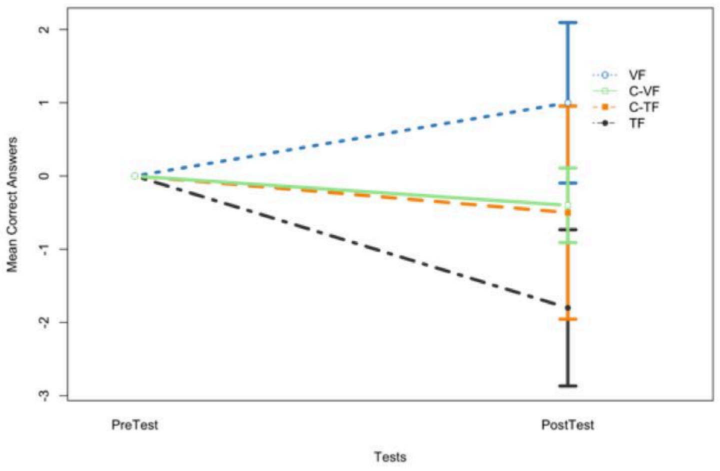


Figure 6.17: Normalised Results relative to Pre-Test

(Lines show error bounds with confidence interval at 95%)

A2: Translation assessment

The second question asked the user to carry out four translations of Visual blocks to Text, and four translations of Text to Visual blocks (see Appendix B). Participation in the translation task during the pre-test was low: 6 of 21 participants attempted a total of 17/168 translations in the pre-test, and 5 of 21 participants completed 29/168 translations in the post-test. There was no correlation between groupings and participation in either test.

Table 6.5: Translation Answer Distribution

(# Correct Syntax features based on correct syntactic element and position)

	Pre-Test	Post-Test
Participant Answering	6/21	5/21
Questions Answered	17/168	29/168
Correct Syntax Features	69%	73%
Correct Answers	2/17	9/29

Each answer in the translation question consisted of between 2 and 7 syntax elements. Scores were given for each answer depending on the number of correct syntactic elements included. Of translations attempted in the pre-test, 69% of syntax elements across all questions were correctly identified. This improved to 73% in the post-test.

Participation in the translation question was sufficiently low that results of statistical tests would have limited statistical power. However, Figure 6.18 shows the three participants who completed the translation question in both tests.

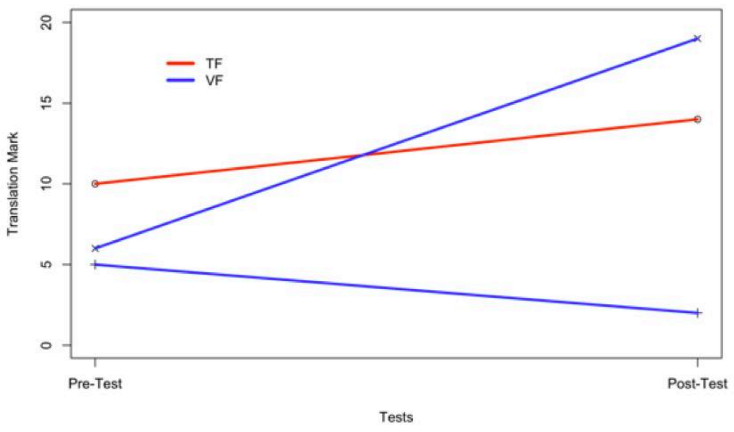


Figure 6.18: Translation Ability by Participating Users

Instrumentation Results

DrawBridge was instrumented to record the actions of participants. Predictably, without-concrete groups spent significantly more time viewing symbolic representations than groups using all representations: without-concrete groups spent 38 minutes 35 seconds viewing symbolic representations, and with-concrete groups spent 3 minutes 23 seconds on average (see Figure 6.19), $F(1,19(= 42.37, p < .0001$.

A Spearman correlation test found a positive correlation between the time participants spent viewing symbolic representations, and the number of syntax errors they encountered $r_s = 0.60, p < .05$. An unpaired t-test found that groups with concrete representations had significantly fewer syntax errors than those without concrete representations $t(9.86(= 5.70, p < 0.001$ (see Figure 6.20).

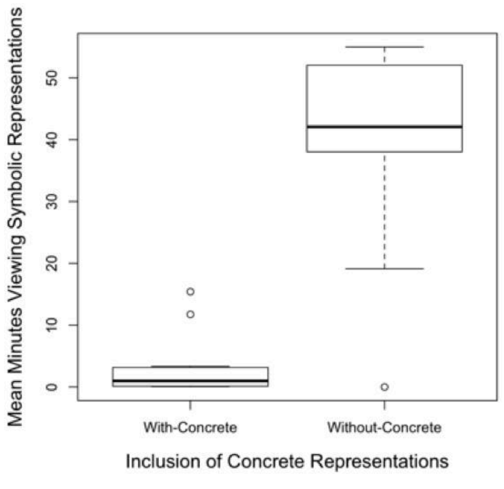


Figure 6.19: Time Spent Viewing Symbolic Representations

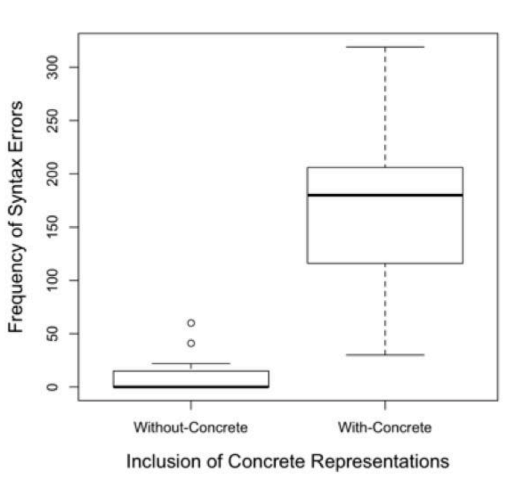


Figure 6.20: Syntax Errors During the Study

Measures of Confidence and Repeated Answers

Data analysis identified that some participants may have been "satisficing" by using a "repeated answers" strategy when answering A1 in the pre and post-test. Ten participants answered "Not Sure" for 8 or more questions of out of 10, indicating that those participants may have had low levels of confidence. When compared with participants' computing skill and enjoyment rating, a Kruskal-Wallis test of reported skill and pre-test repeated answers showed a non-significant interaction $\chi^2(n = 21, 4) = 7.40$, p = 0.12. A metric for repeated answers was created to identify participants with low confidence (**Equation 1**), following previous research (McCarty & Shrum, 2000). The metric, shown in **Equation 1**, calculates the difference between the minimum and maximum number of the same answer (a = "Yes", b = "No", c = "Not Sure"}.

$$D = \frac{\max\left(\sum a\ ,\sum b\ ,\sum c\ \left(-\ \min\left(\sum a\ ,\sum b\ ,\sum c\ \left(\right.\right.\right.\right.}{n}$$ **Equation 1**

Confidence can be measured directly using the frequency of "Not sure" answers for each question in the pre-test and post-test. Figure 6.15 shows the relative percentage of "Not sure" answers for each group. There was a non-significant decrease in confidence levels in all groups apart from C-VF between the pre-test and post-test.

6.5.2 Lesson 2

The non-equivalent grouping encountered in Group TF during the syntax assessment stimulated a second visit to the same class. In the second lesson, all participants were given the “With-concrete and Visual-First” (C-VF), version of DrawBridge. Results from Lesson 2 will be presented using the group names given in Lesson 1, to distinguish between participants based on experience in Lesson 1. For example, Group “TF” used the TF version of DrawBridge in Lesson 1, and C-VF in Lesson 2.

A1: Syntax Assessment

A1 in the Lesson 2 post-test had a high level of participation (98.89%), improving from the previous visit. The two “starter” questions were answered with 94% participation.

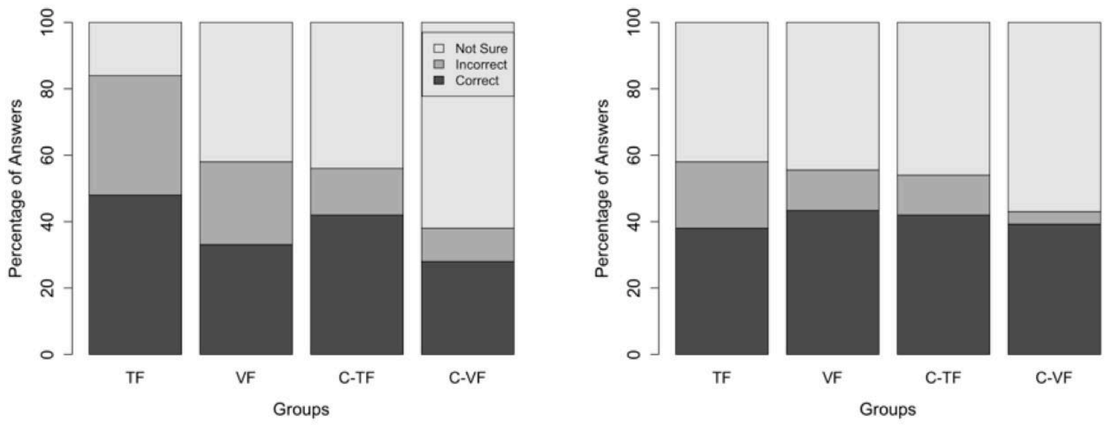


Figure 6.21: Lesson 2 Answer Categorisation by Group
Left: Lesson 1 Post-Test **Right**: Lesson 2 Post-Test

All groups, with the exception of Group TF, improved between the Lesson 1 post-test and Lesson 2 post-test: on average VF improved by 1 (SD=0.70), C-VF improved by 1 (SD=1.73), C-TF (SD=3.78) improved by 0.6. The further decline of Group TF’s results (mean = -1, SD=1.22) could have been due to a regression towards the mean (see Figure 6.22). Group C-TF (the Text-First, with-concrete group in Lesson 1) showed the most improvement.

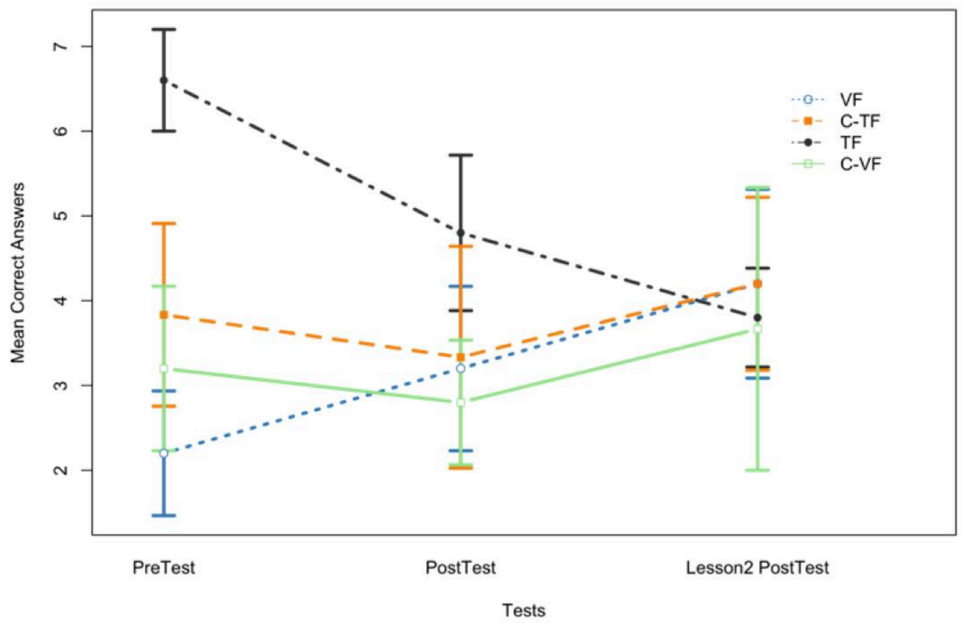


Figure 6.22: Absolute Syntax Assessment Results

An analysis of variance (ANOVA) showed that there was no significant difference between participant performance in the Lesson 1 post-test and Lesson 2 post-test: $F(31) = 0.408, p = 0.527$. There was also no significant difference between group performance between the Lesson 1 post-test and Lesson 2 post-test: $F(31) = 0.349, p = 0.790$,

A2: Translation Assessment

In the Lesson 2 post-test, students were asked to make a special effort to try to answer A2. Participation increased to 58% (from 17.3% in the Lesson 1 post-test). 49% of syntax elements across questions were correctly identified.

Table 6.6: Translation Results

	Pre-Test	**L1 Post-Test**	**L2 Post-Test**
Participant Participation	6/21	5/21	14/18
Question Participation	10.11%	17.26%	57.63%
Correct Syntax Features	69%	73%	49%
Fully Correct Answers	2	9	14

The low level of A2 participation in the Lesson 1 post-test made it only possible to analyse the change in three participants' answers. Figure 6.23 shows that participants' translation ability improved the more time they spent using DrawBridge.

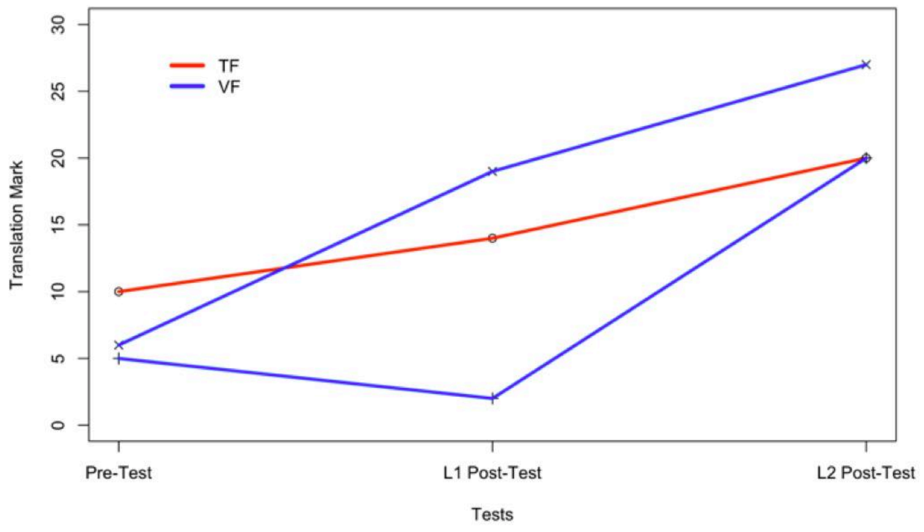


Figure 6.23: Translation Ability Between Tests

DrawBridge Instrumentation

The time spent viewing symbolic representations improved from 3 minutes 23 seconds on average in Lesson 1, to 15 minutes 25 seconds in Lesson 2.

Confidence Levels

Despite a general increase in confidence levels, which can be seen from the proportion of "Not Sure" answers in Figure 6.21, there was no significant difference between the number of "Not Sure" answers in the Lesson 1 post-test and the Lesson 2 post-test. Group C-VF had the highest confidence on average in Lesson 2, suggesting that confidence might increase as familiarity with the environment increases.

A Kruskal-Wallis test found that reported motivation to use the system again in the post-questionnaire varied significantly with the change in student confidence between the start of Lesson 1 and end of Lesson 2, $\chi^2(n = 21, 3) = 7.965, p = 0.047$. Curiously, participants responding that they would "Definitely" use DrawBridge again showed reduced confidence (4 more "not sure" answers on average), and participants responding "Not Really" to using DrawBridge again showed increased confidence (6 fewer "not sure" answers on average). It is possible that students who had high confidence did not feel they benefitted from using the system.

Motivation Questionnaire

The motivation questionnaire was included to provide an additional measure of participants' confidence. Results from motivation Likert scales show that most participants enjoyed using DrawBridge, and would like to use it again (Figure 6.24), suggesting that their self-efficacy had become sufficiently high to continue using the system.

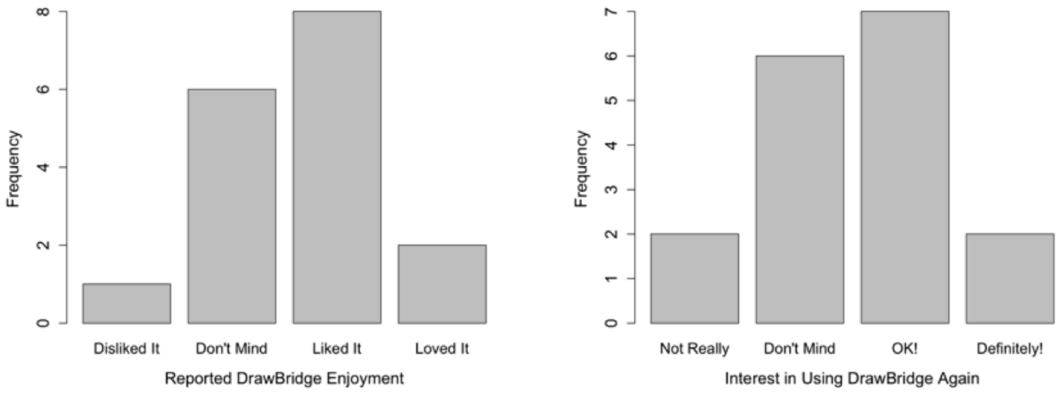


Figure 6.24: Participant Feedback after using DrawBridge

Left: Reported Enjoyment after Lesson 2 Post-Test, **Right**: Reported Interest in Further Sessions

The final motivation questions related to the three best things and three worst things about DrawBridge were coded into categories, shown in Table 6.7 and Table 6.8. Animating with DrawBridge was the most frequently mentioned "best" thing, closely followed users drawing their own characters. Participants enjoyed "seeing the characters on the screen" and the fact one could "see your own animations. It's fun! It's different!"

Category	Frequency
Animating	13
Drawing characters	11
Watching Animations	6
Seeing Programming Code	1

Table 6.7: Coding of the "Best Things" about DrawBridge

Category	Frequency
Unreliable	9
Complicated	5
Confusing	4

Table 6.8: Coding of the "Worst Things" in DrawBridge

During the study, some participants experienced reliability issues with DrawBridge, stating that “sometimes it froze”, and “It’s a bit slow sometimes”. Other remarks were made in relation to DrawBridge being confusing: “Sometimes it’s confusing” and “Some of it is confusing”. Other participants mentioned “All the text that you don’t understand” and “The code looked difficult” as being some of the worst things about DrawBridge.

Analysis found that users who answered less enthusiastically had a have a higher repeated answering metric on average, based on the Lesson 2 post-test (see below).

Table **6.9**: Repeated Answers Metric and Motivation to use DrawBridge

Use Again?	Lesson 2 Post-Test Repeated Answers (Higher value indicates more repeated answers)
Definitely	0.4
OK	0.37
Don’t Mind	0.63
Not Really	0.67

6.6 Discussion

The purpose of this study was to explore the relative benefits of correspondences between pairs of representations in MER systems, including pairs originally generated from the MoRA framework with the intention of maximising student acquisition of Notational Expertise (NE).

This section will discuss the results from Lesson 1 and Lesson 2 of the study in the context of the questions presented in Section 6.4.

6.6.1 Participant Background and Self-selection Bias

Responses to the questionnaire given in Lesson 1, and observation during the lesson indicated that students were uncertain of the definition of “programming”. The majority either thought programming was exclusively text-based, or did not have enough experience with programming to understand what they were doing. The latter explanation appears to be more likely given that very few participants mentioned having used text-based programming. This result is consistent with Lewis et al., who found that the term “programming language” was not central to classification of Scratch (Lewis, Esper, & Bhattacharyya, 2014). Despite their confusion, participants reported encouraging levels of computing enjoyment and skill before the study began.

The group results in the pre-test A1 showed that group TF performed significantly better than any other group, suggesting that there may have been an implicit self-selection bias due to participants seating positions. The self-selection bias is likely to have been due to students sitting with friends, who may have had similar levels of academic or programming ability. Students from TF reported their own skill as “Good”, “Very Good” and “Excellent”, suggesting a high level of self-efficacy from the outset. Interestingly, TF was the only group to contain participants reporting as “Excellent”. Answers given by participants in Group TF were sufficiently distributed to suggest that there was no collusion between students, and only three of five of TF participants reported to having programmed before. To reduce the impact of self-selection bias in Group TF, change scores were used during analysis of results (see Figure 6.17). This method follows previous research (Levy et al., 2003) and best practice (May, 2012). Mackinnon et al found that across four studies, students would sit with others that were physically similar to themselves (Mackinnon, Jordan, & Wilson, 2011). The self-selection bias in this study may be indicative of classroom behaviour in which students with similar academic goals or ability sit together.

6.6.2 Including Concrete Representations

The first question presented at the beginning of this chapter questioned the extent to which including concrete representations (Paper, Direct Manipulation and Abstract DM), with animation features, might improve students’ acquisition of Notational Expertise.

Results show that there was no significant difference between groups with and without concrete representations between the pre-test and post-test. However, results from the motivation questionnaire in Lesson 2 suggest that concrete representations were a major motivation for using DrawBridge. Instrumentation data collected in Lesson 1 and Lesson 2 shows that with-concrete groups used symbolic representations a lot less, providing some explanation for their poorer performance.

Analysis of Lesson 2 results suggest that on average, participants with longer exposure to the visual-first version of DrawBridge (those that were in Group VF and C-VF in the initial visit), improved more than other participants, although it is not known how participants would have fared given a second lesson with other versions of DrawBridge used in Lesson 1.

The A2 translation task had very low participation, which limits the strength of any conclusions drawn from comparison of A2 in the pre-test and post-test. Three participants completed A2 in the pre-test and post-test. Four participants completed A2 in both the Lesson 2 post-test and the Lesson 1 post-test. The increase of translation ability in participants who responded to A2 was encouraging. However, participants who responded were from VF and TF groups; it is therefore not possible to comment on the translation ability of with-concrete participants. The response was the same in the pre-test so participation is unlikely to be due to grouping.

Motivation questionnaire results were extremely positive towards the use of pen and paper. Almost all participants mentioned the use of paper and concrete representations as one of the best things about DrawBridge. These results agree with previous findings in the pilot study (see Section 6.3) and teaching interviews presented in Chapter 4.

6.6.3 Order of Symbolic Representations and Notational Expertise

The second question explored in this chapter asked the extent to which participants using the visual block representation before the text representation would acquire more Notational Expertise than participants using the text representation before the visual block representation. To explore this question, in Lesson 1 of the study, the participants were given customised versions of DrawBridge with reversed orders of symbolic representation (Visual-First and Text-First). The change in order of symbolic representation results in different pairings of representations in DrawBridge, and therefore different usability characteristics and types of correspondence for each pair. For example, in Text-First, correspondences in the DM-text pair highlight the meaning of parameters in the context of text syntax; in Visual-First, the DM-visual block pair highlights correspondences between objects and number blocks, which do not show the syntax features that will be needed to express the change in text-code.

Analysis of results from Lesson 1 found that groups using the visual-first versions of DrawBridge improved more than those using text-first versions. Although this difference was not significant, it suggests that there may be some benefit in using visual representations before text representations for students. Within visual-first groups, VF improved more than C-VF on average; both groups continued to improve when using C-VF in Lesson 2.

Instrumentation analysis suggested that the motivation provided by low-abstraction concrete representations may have been distracting, stopping participants from making the transition to subsequent symbolic representations. Results show that participants in “with-concrete” groups spent significantly less time using symbolic representations compared to participants in “without-concrete” groups, with C-VF using 12% of the time VF did, and C-TF using 8% of the time TF did. Furthermore, students may have been restricted due to the short time available during the lesson. The increase in time spent on symbolic representations in Lesson 2 suggests both may be true; participants are distracted by animations, but also need more than a single lesson to make the transition to symbolic representations. This result appears to support Cockburn and Bryant’s analysis of Leogo, which found that students typically used the same representation for a complete task and avoided using the text representation (Cockburn & Bryant, 1997).

All participants in Lesson 2 were given the C-VF version of DrawBridge due to change scores for visual-first groups being higher than change scores for text-first. The majority of participants showed further improvement in Lesson 2. However, students from Group TF in Lesson 1 did not improve on average, suggesting that participants either continued to regress towards the mean of the entire class, or that they may have been confused or demotivated by using a text representation first.

6.6.4 Ordering and Concrete Representations

The third question explored in this chapter asked whether the inclusion of low-abstraction concrete representations would have an impact on gain scores with students using different orders of symbolic representations. There was no significant interaction in the change scores of participants between the two factors: ordering and the inclusion of concrete representations.

Analysis of instrumentation data shows that participants with concrete representations spent significantly less time using symbolic representations in general, perhaps explaining why groups without concrete representations performed better on average in the Lesson 1 post-test than groups with concrete representations. It was not possible to measure enjoyment and motivation of individual groups as the motivation questionnaire was given as part of the Lesson 2 post-test, which was distributed after all participants had used the C-VF version of DrawBridge.

The motivation questionnaire given in the post-test of Lesson 2 confirmed that concrete representations provide a major motivation for users of DrawBridge, and that the most users liked the system and would be happy to use it again.

6.6.5 Syntax Learning Implications

The pre-test and post-test comparisons support the assertion that a visual-first ordering of representations is preferable to a text-first ordering. Although there was no significant difference between groups that used concrete representations and groups that did not, feedback in the Lesson 2 questionnaire show that the use of concrete representations provide major motivation to students. Furthermore, DrawBridge instrumentation data suggests that participants with concrete representations had less time to use symbolic representations, and therefore could not have performed as well as without-concrete groups.

6.6.6 Confidence Levels

Analysis of pre-study questionnaire results show that reported skill corresponded with performance in the pre-test (see Section 6.5.1). Participants reporting higher skill may have had more confidence due to more experience with computers. However, all students had similar programming experience (All had used Light-Bot, most had used Scratch), suggesting that other factors may have influenced participant confidence.

The “Not Sure” option in A1 of each assessment was introduced to measure confidence. The overall number of “Not Sure” answers fell between the pre-test and the post-test, suggesting that participant confidence improved. However, the increase in confidence did not result in a direct increase in correct answers: Group TF responded with low numbers of “Not Sure” answers in the pre-test and post-test, but had a large increase in incorrect answers, suggesting that participants had reduced efficacy but may not have been aware of it.

A post-hoc analysis of confidence and questionnaire feedback shows that the decision as to whether participants would use DrawBridge again was related to their change in confidence during the study.

The “starter” questions added to A1 to increase confidence were answered correctly 94% of the time in the pre-test, and 70% of the time in the post-test, despite having identical syntax. The fall in correctness can be attributed to a fall in confidence, as the total number of “Not Sure” answers increased from 1 to 8. The responses improved to 94% in the Lesson 2 post-test, with the number of "Not Sure" answers dropping to 2, suggesting that confidence may return to its original level over time.

Finally, the repeated-answers differentiation metric, also used as a measure of confidence, suggested that participants' interest in using DrawBridge again varied depending on the level of satisficing participants used, measured by the number of repeated answers participants given. Satisficing may have been due to low enjoyment or motivation.

6.7 Follow-up Think-Aloud Study

A follow up study was carried out with three new participants aged 11-12 using the concurrent think-aloud protocol to investigate unexpectedly high results from Group TF in the pre-test, and poor user participation from all groups in A2 on the pre-test and post-test. The study was carried out with participants on a one-on-one basis, either at their home or in the Usability Laboratory of the Computer Laboratory in Cambridge.

Two female participants completed the study using the C-VF version of DrawBridge, while one male participant used the C-TF version. In addition to methods of existing data collection, participants' comments during think-aloud studies were recorded using a high quality microphone and transcribed.

6.7.1 Protocol

At the start of each study, students were given a tic-tac-toe game to play with the researcher, who demonstrated the think-aloud protocol and encouraged the participant to practice while playing. The protocol for the main part of the study was consistent with the protocol used in schools. However, users were prompted to think aloud if they forgotten to do so.

6.7.2 Results

The quantitative results recorded for these participants were comparable to the performance of participants in the main study. However, these results primarily focus on qualitative results gathered from participant comments transcribed from audio recordings made during the study. Transcripts were coded using a mixture of pre-defined and emergent codes. Codes were then collected and grouped using the similarities and differences method described in Chapter 4.

Pre-Questionnaire

As in the main study, there was a mix of experience with animation and webpages. All three participants had used Scratch and SmallBASIC, and described their computing enjoyment as "it's ok" to "Love it", and computing skill as "Fair" to "Good".

A1: Syntax Assessment

The most significant finding during the think-aloud study was that students were returning to previous questions to develop answers to new questions. One student even turned back to the previous page to check answers she had given:

> *"Actually maybe it does have brackets for those…in which case that wouldn't be right [turns over] that means that one wouldn't be right, and that one. Which would mean that none of the code questions were right…so actually maybe it's not like that."*

Another participant used a similar technique: "Which means that…it's right. And then that is right, and that one is right, actually no that one isn't right because of the variable things". The third participant stated "Because I had a look at these numbers in this one. I should probably go back. Can I go back?"

One student found it difficult to interpret the wording of the question "Do these pieces of text look correct?", stating "By correct does it mean like…what does it mean by correct?" The two "easy" starter questions, intended to give students a familiar email and URL to increase confidence at the start of A1 also added to their confusion "So whether [the starter questions] would work in the system or as a web address or email?"

Another participant found it difficult to differentiate the questions themselves from previous work he had done in mathematics "So $x = 2 + 2$, so $x = 4$!". During their use of DrawBridge, only one participant realised that RGB stood for Red, Green and Blue. All participants had to be told that the parameters for Red, Green and Blue were integers between 0 and 255. These problems are due to lack of supporting knowledge, which teachers highlighted during the interviews discussed in Chapter 4.

Participants understood that the renamed function "setTimeToDestination" meant the time in between animation steps. However, they were confused with the word "tween", which was still used as a method name to set the destination position and size of each image: "What does tween mean?"

Comparison of Question Difficulty

Participants had divided opinions as to which task was the most difficult. The first participant stated that the A2 was easier, although admitted that in A1 "there's only three choices so you could probably write anything". The second stated that both questions "...were equally hard".

The third participant stated that the A1 was easier than the translation task because “you just had to say whether you thought it was right or not”.

6.7.3 Discussion

The think-aloud study was conducted to further elucidate results collected in Lesson 1 and 2 of the main study. The main finding of the study was that participants attempted to guess the correct answers to A1 by making inferences from common syntax in other parts of the questions, possibly explaining the high pre-test results achieved by Group TF.

The study also found that students had a lack of supporting knowledge in programming contexts. Despite being explained in the accompanying worksheet, coordinates, animation terminology, and RGB values had all been misunderstood in DrawBridge.

Finally, although design decisions in DrawBridge had explicitly attempted to address confounding similarities between mathematics and programming, students found it difficult to reconcile their existing mathematical understanding of equations with new programming statements, in which the “=” sign has a different meaning.

Limitations

The think-aloud study showed that it was possible to use later questions in the assessment itself to infer correct answers for earlier questions. However, only a subset of questions could be inferred correctly, and results for other groups in the pre-test suggest that the majority of participants were not using this strategy.

6.8 Design Implications for DrawBridge and Other MERs

The results gathered from think aloud studies and both classroom experiments, as well as informal observations made during the studies, led to several design recommendations for the DrawBridge system, and more broadly, for the design of future MER systems.

Drawing on Paper as Motivation

As described earlier, feedback for the mechanism of providing motivation to students using DrawBridge – capturing and animating characters drawn on paper – was extremely positive. The range of drawings, character types, use of colour, and experimentation observed during these studies, in addition to positive questionnaire feedback, demonstrated the students’ desire for self-expression, customisation and personalisation during programming. The ownership of characters taken by students during the experiment was clear to see, and appeared to provide a good basis on which to learn about programming. Instrumentation and observations showed that this design decision was possibly too successful, as students focused more on characters and animation than programming. However, custom hand-drawn characters will remain the major source of motivation for students using DrawBridge in the future. Existing novice programming systems provide various sources of motivation, as described in Chapter 2. As with the design of DrawBridge, future MER systems should endeavour to provide starting points that are equally motivating for both males and females, and attempt to provide a concrete base that students can use to reify the abstract concepts of programming. Ideally this motivation would not distract the students from interacting with the programming aspects of the system.

Panorama as a Representation Transition Metaphor

DrawBridge was designed to use a panorama metaphor with representations identified using the MoRA framework, presented in Chapter 3, and refined using the CDs framework, described in Chapter 5. Although the panorama metaphor appeared to help students identify the appropriate order with which to interact with DrawBridge representations, they often first scrolled through the representations before returning back to the start to begin interacting with the system. Some students, in particular those with programming experience, became competitive regarding the speed with which they could use the system and move ahead of the rest of the class.

Cognitive Dimensions as a Design Tool

The CDs framework proved to be useful in identifying the usability characteristics of each representation, and representation pair, that might introduce difficulties for students. For example, the increase in abstraction and hidden dependencies between the Abstract DM panel and Visual Block panel caused students to run into errors when manipulating blocks. The system supported these problems, by allowing students to recover from them by "resetting" their DM panel. The real-time feedback also appeared to help students to identify errors.

Despite an overall recommendation for the use of the CDs framework when designing new MER systems, the usability of at least two dimensions of the CDs proved to be difficult to predict with students using the DrawBridge system. For example, High *visibility*$^{(CD)}$ of a representation is generally described by CD literature as a positive attribute. However, observations of students using the text-based representation suggested that the increased visibility may have increased cognitive load, and been intimidating for novices. The *role expressiveness*$^{(CD)}$ of each representation was also difficult to predict accurately as students using MER systems were able to infer the role of notation from the notation itself, but also notation emphasised using correspondence highlighting.

Lack of Supporting Knowledge

Observations and think-aloud feedback identified that students found some parts of DrawBridge confusing due to a lack of supporting knowledge – knowledge required to use the system that is not directly related to programming. Despite attempts being made to reduce this, supporting knowledge was needed in DrawBridge to specify colour and time for animation steps. Although it may be impossible to remove it completely, further reduction of the required supporting knowledge is likely to improve the user experience for novice programmers. If removal is not possible, features requiring supporting knowledge should be described to students using supporting documentation or demos to allow them to focus on the programming elements of the system rather than domain-specific issues.

6.9 Conclusions

The purpose of this study was to explore the validity of strategies generated using the MoRA framework, and to support further design decisions by investigating alternative pairs of representations which might improve student acquisition of Notational Expertise. By exploiting the flexibility built in to DrawBridge, four unique versions of the system were created to study the effect of changes in the order of symbolic representations and the inclusion of concrete representations on student acquisition of Notational Expertise.

Findings show that visual-first groups improved more than text-first groups, and kept improving over multiple lessons, suggesting that a visual-first order would be the most appropriate order for novice programmers using DrawBridge. The results also showed that the use of text as an introductory representation might reduce participant confidence and provide no distinguishable improvement in competence, which is undesirable.

Overall, there was no significant difference between groups with and without concrete representations. The small difference in mean gain is likely to be due to participants with concrete representations spending less time viewing symbolic representations, which itself confirms a previous result reported by Cockburn and Bryant (Cockburn & Bryant, 1997). Despite no significant difference in assessment scores, questionnaire results and think-aloud feedback suggest that concrete representations do provide major motivation for participants, who stated that they particularly enjoyed drawing and creating animations. Results from both lessons show that participants using visual-first versions of DrawBridge continued to improve with extended use.

A complementary think-aloud study identified that the most likely cause of high pre-test results for Group TF was a weakness in the design of the syntax assessment; participants were returning to previous questions to infer the correct answers despite having low levels of Notational Expertise.

Four implications for the design of DrawBridge and other MERs were described based on the observations and analysis presented in this chapter. The implications related to the positive feedback based on the use of paper and hand-drawn characters, modifications to the panorama navigation metaphor used for representation transition, reflection on using CDs as a design tool, and minimising the requisite supporting knowledge.

In summary, these quasi-experiments address the important issue of pairings of representations, and more specifically, the order and inclusion of concrete and symbolic representations in MER systems used in a classroom context. Although results from the use of different versions of DrawBridge over a single lesson did not yield significant differences, they suggest that the use of visual blocks to scaffold student knowledge before introducing them to text representations may improve acquisition of Notational Expertise. Further, a novel method of providing motivation for programming systems was identified.

The challenges of designing and administering effective assessment methods for measuring Notational Expertise resulted in low response rates for translation questions, and limitations in the results gathered from the first question. Several questions arose from this; principally, can syntax assessments be designed such that participants cannot easily infer and answer questions without sufficient Notational Expertise, and can syntax assessments be designed to elicit high response rates, low repeated answer rates, and an increased understanding of students' mental model of syntax structure? These questions are addressed in the next chapter.

Chapter 7 Designing Assessments for Notational Expertise

7.1 Introduction

The previous chapter presented several quasi-experimental studies investigating the acquisition of Notational Expertise (NE) using DrawBridge, a novel educational programming environment using Multiple External Representations (MERs). In the studies, a simple two-part pre/post-test was developed to assess improvement in NE as a result of using DrawBridge. Unexpectedly, analysis of assessment results found significant differences between groups of students taking the pre-test. A follow-up think-aloud study, designed to investigate these differences, suggested that the design of the assessment was problematic, and that some students may have used later questions to inform their reasoning for answers to earlier questions.

In response to these issues, and to calls for assessment methods of novice programming environments to be made more rigorous (Gross & Powers, 2005), this chapter aims to investigate the following research question: RQ4: Which type of assessment would be most appropriate for measuring Notational Expertise? To answer this question, an investigation was carried out to compare four candidate assessment types identified in the computer science education literature: Code Writing, Debugging, Multiple-Choice Question (MCQ) and Adapted Parsons Problems. Each assessment used the same set of questions relating to NE, and was developed to be usable by researchers and teachers. The most suitable assessment, Adapted Parsons Problems, is used in later chapters to measure improvement in NE.

The structure of this chapter is as follows. Section 7.2 discusses assessment findings from teaching interviews conducted in Chapter 4, followed by a review of related literature on novice programming assessment. Section 7.3 presents the designs of four candidate assessments that are compared in the subsequent study. Section 7.4 describes Code Kingdoms, a game-based novice programming environment used during the workshop by all students before taking the assessments. Section 7.5 presents the design of the study, followed by the results in section 7.6 and discussion in section 7.7. Finally, a chapter summary is given in section 7.8.

7.2 Related Work

Assessments are used to gain information about student achievement; they can be used by teachers or researchers to adjust content and the delivery of instruction by answering questions such as "To what extent are students attaining the learning goals of this course?", "What types of learning difficulties are students encountering?" and "How effective was this intervention?" (Linn, 1995). This section will discuss existing assessments used to measure student achievement using novice programming tools in primary and secondary school, and beyond, at undergraduate level. These assessments will be used to create candidate assessments for the measurement of Notational Expertise in future studies.

What Kinds of Assessment are Currently Used in School?

In the United Kingdom, programming assessments in secondary education are often given in the form of summative assessments, or exams, that take place at GCSE (age 16) and A-level (age 17-18). Analysis from semi-structured interviews conducted with teachers presented in Chapter 4 suggests that while programming assessments are also used formatively[1] in classrooms, they are given infrequently and informally as part of a wider class discussion. Interviews also suggest that in addition to testing, teachers implicitly assess student progress by moving around the classroom, or by listening to feedback and the types of questions asked.

However, more rigorous formative assessments may be desirable; teachers who use regular formative assessments with Multiple Choice Questions (MCQs) and code writing questions before exams, and use the results of formative assessments to determine student misconceptions, successfully increase student intake and pass rates in high school courses (Ericson, Guzdial, & Mcklin, 2014).

What Kinds of Assessments do Researchers Use?

The goals of technology researchers are likely to be different to the goals of teachers; technology researchers develop assessment instruments to evaluate the success of tools or interventions intended to improve programming knowledge. However, despite good intentions, there has been criticism of the way in which assessments are developed and reused by computer science researchers investigating novice programming tools. Gross and Powers

[1] Despite some ambiguity in the term "formative assessment" (Black & Wiliam, 1998), for the purposes of this thesis, it will be used to refer to any assessment tool that is used within the learning process to provide valuable feedback to both students and teachers, and guide future learning.

suggest that there has been too little rigorous assessment of such tools, which has limited understanding of their impact (Gross & Powers, 2005). An evaluative framework provided by Gross and Powers based on five representative assessments of novice programming environments suggests that future research should be more problem-based, rather than opportunity-based, and be fully documented in order to support replication. In addition to these deficiencies, a number of assessments are completed in non-classroom contexts, such as summer camps (Ericson & McKlin, 2012; Esper, Wood, et al., 2014; Kelleher, 2006) and after-school computer clubhouses (Maloney et al., 2008), potentially reducing the validity of results when considered in a classroom context, where activities and assessments are led by teachers, and emphasis is placed on achieving learning goals.

Many types of assessment have been used to measure student performance using novice programming environments in primary and secondary schools. Each type can be used to test a variety of cognitive processes, described in the revised version of Bloom's Taxonomy (Krathwohl, 2002; Thompson, Luxton-Reilly, Whalley, Hu, & Robbins, 2008). One commonly used method is to examine the artefacts created by students who have used the system over a fixed period of time (Maloney et al., 2008). This type of assessment is limited, as it can be influenced by collaboration, plagiarism, or trial and error. A second type of assessment is to provide students with a piece of code, and ask them to perform modification or debugging tasks (Esper, Wood, et al., 2014; Lee et al., 2013). This type of assessment can be useful in determining whether students understand particular syntactic or conceptual concepts, and limits the complexity of the cognitive processes students must access (e.g. "creation" and "evaluation" processes in the revised version of Bloom's Taxonomy (Krathwohl, 2002)). A third type of assessment is to ask students to select the correct answer from a set of possible answers in MCQ assessments (Ericson & McKlin, 2012). Researchers have also used heterogeneous assessments, which can include several question types, such as code writing, terminology identification, and MCQ (Esper, Foster, Griswold, Herrera, & Snyder, 2014; Levy et al., 2003).

Undergraduate Programming Assessment

The majority of programming assessments reported in the computer science education literature are given in undergraduate level courses, where content and assessments can be improved iteratively each year. Methods of assessment at this level include program writing, explain in plain English (Murphy, Fitzgerald, Lister, & McCauley, 2012), predict the output (Mishra, 2014), tracing, MCQ and debugging. An investigation into the common characteristics of CS1-level introductory programming examinations found that the majority of marks were allocated to code writing questions (Simon et al., 2012), with less than 36% allocated to short-answer questions and less than 10% for MCQ and others. This distribution may change in the future; Woodford suggests that despite previous criticism of MCQ tests from researchers who suggested they could only be useful for assessing superficial memorising of facts, increasing numbers of CS1 lecturers believe that they can be used to test deeper levels of understanding, if designed correctly (Woodford & Bancroft, 2005).

Researchers have frequently studied the relationship between code writing and code tracing. Teague and Lister found that the majority of the CS1-level students in their 303-participant undergraduate study could answer tracing questions, but could not explain what the program they were tracing did, suggesting that they would not be able to write similar code (Teague & Lister, 2014). There have been suggestions that code tracing may act as a pre-cursor skill to code writing (Lister et al., 2004).

In addition to summative exam questions, formative assessments, such as answering MCQs using clickers (devices with which students can anonymously give their answer), have been used in conjunction with peer instruction methods during lectures to improve student understanding, discussion and engagement with the assessment process (Simon et al., 2010).

Parsons Problems

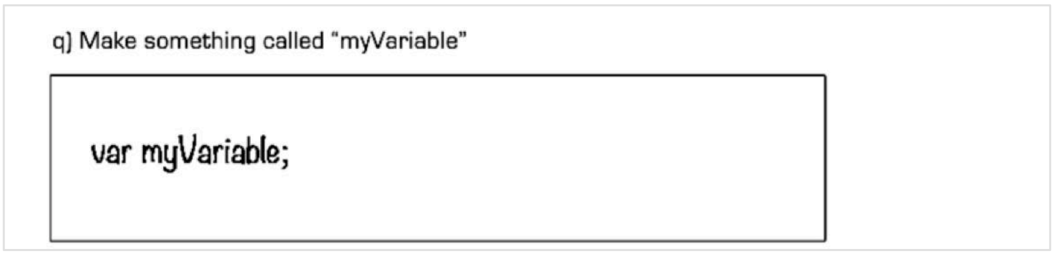
Parsons programming puzzles were introduced by Dale Parsons in 2006 to increase student engagement with programming activities by providing a puzzle in which they can drag and drop fully formed lines of code into position to complete the required program (Parsons & Haden, 2006). Denny et al. proposed that Parsons Problems could be used as a new type of exam question which could improve upon code-tracing, which is disliked by students, and code writing, which is difficult to mark (Denny, Luxton-Reilly, & Simon, 2008). Denny's analysis of results from a CS1 final exam using Parsons Problems showed that results correlated with code writing results, suggesting that the assessment measures the same skills, while being easier and more reliable to mark.

7.3 Assessment Design

Four programming assessments were selected from those reviewed in the previous section as candidate assessments for the measurement of Notational Expertise. Each assessment type was designed to be paper-based and to contain nine questions that would be consistent across all assessments. An example question was provided at the start of each assessment with an answer appropriate to the assessment type. Each question addressed a different text syntax feature, as described in Section 7.5.1 and provided in full in Appendix C.

Code Writing Assessment Items

Code writing assessments typically require students to create partial or complete programs that would complete a particular task. They require the respondent to use "create" cognitive processes, which are considered to be the most complex in the revised version of Bloom's Taxonomy (Krathwohl, 2002), and thus are expected to be more difficult than questions requiring less complex cognitive processes, such as those asking the student to debug a given program.


q) Make something called "myVariable"

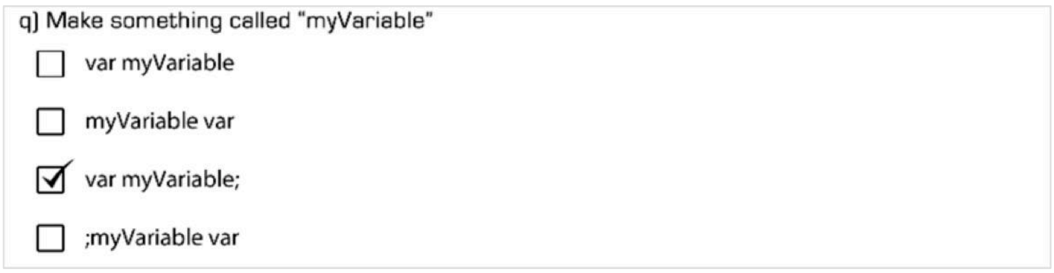
```
var myVariable;
```

Figure 7.1: Example Code Writing Assessment Item

Multiple Choice Assessment Items

Multiple-choice questions (MCQs) allow respondents to pick from a number of answers, of which just one is typically correct. They are commonly used to measure knowledge of terminology, specific facts and principles (Linn, 1995), and are becoming an increasingly accepted type of assessment within computer science education (Woodford & Bancroft, 2005).

Designing MCQs for measuring Notational Expertise is not trivial. Incorrect answers, or "distractors", should be plausible alternatives that respondents would select had they not achieved the required learning outcomes (Linn, 1995). However, the limited number of syntax elements, and the requirement for context within each answer allows respondents to apply shallow heuristics to infer the correct answer without necessarily achieving the desired learning outcome, repeating the issues identified with A1 in the assessment given in the previous chapter.

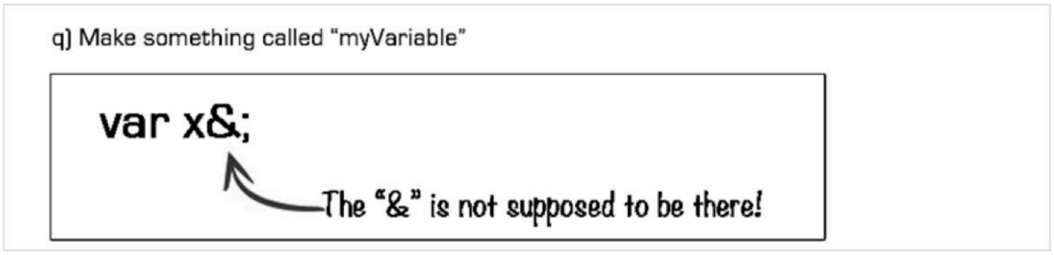

q) Make something called "myVariable"

- ☐ var myVariable
- ☐ myVariable var
- ☑ var myVariable;
- ☐ ;myVariable var

Figure 7.2: Example MCQ Assessment Item

Debugging Assessment Items

Debugging items provide the answer to each question with one or more additions, removals or modifications. Students are required to compare a given statement to their own mental model of how statements are constructed. Designing modifications in debugging tasks is comparable to designing distractors in MCQ items; they must be plausible enough to separate students who have and have not achieved the required learning goals – in this case, Notational Expertise.

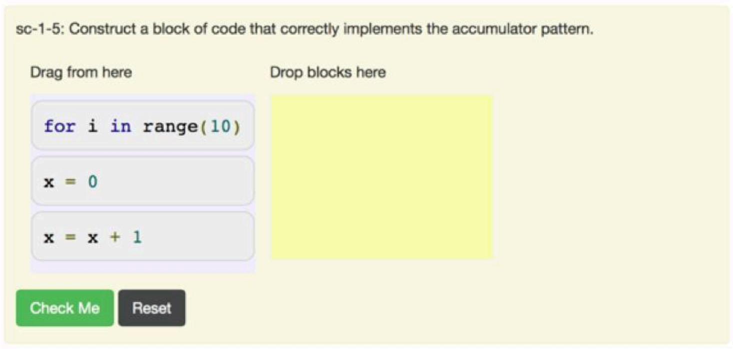

q) Make something called "myVariable"

```
var x&;
```

Figure 7.3: Example Debugging Assessment Item

7.3.1 Adapted Parsons Problems

Parsons Problems appear to be a promising, fun, and engaging method to formatively assess acquisition of Notational Expertise within the classroom. However, existing Parsons Problems, such as those in Figure 7.4, have been created to assess student understanding of whole programs in summative assessments at the end of programming courses.


sc-1-5: Construct a block of code that correctly implements the accumulator pattern.

```
for i in range(10)
```

```
x = 0
```

```
x = x + 1
```

Figure 7.4: Traditional Parsons Problems

In this investigation, Parsons Problems have been adapted to assess knowledge of individual statements rather than whole programs. With these "Adapted Parsons Problems", students are required to position syntax elements in the correct order from left to right instead of positioning lines in the correct order from top to bottom (Figure 7.5).

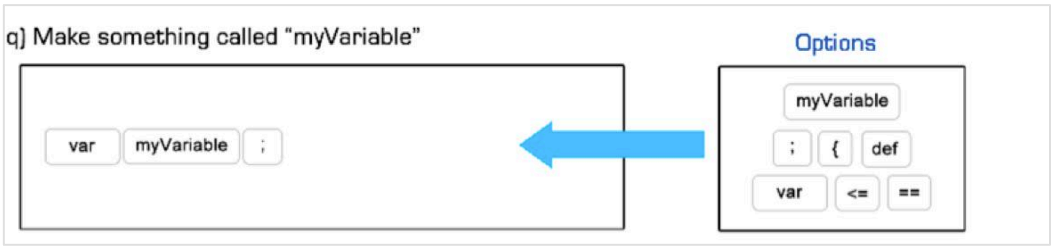

q) Make something called "myVariable"

Figure 7.5: Adapted Parsons Problems

7.4 Code Kingdoms

Code Kingdoms (Code Kingdoms, 2014) (CK) is a recently developed, game-based, novice programming environment in which students create virtual worlds with programmable objects and characters. Users of CK are able control the design of their worlds using a design editor (Figure 7.6), which allows them to manipulate the properties and behaviour of characters, objects and obstacles, which include different terrains such as water or volcanic lava, and enemy characters called "glitches".

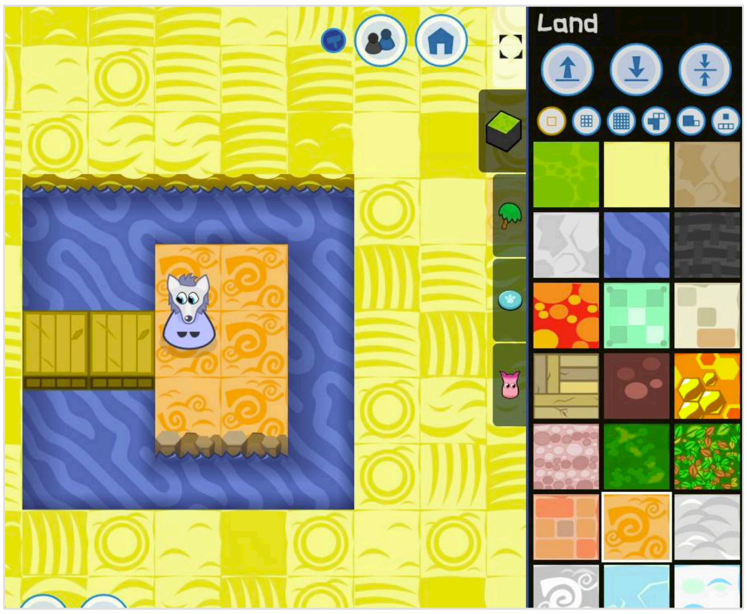
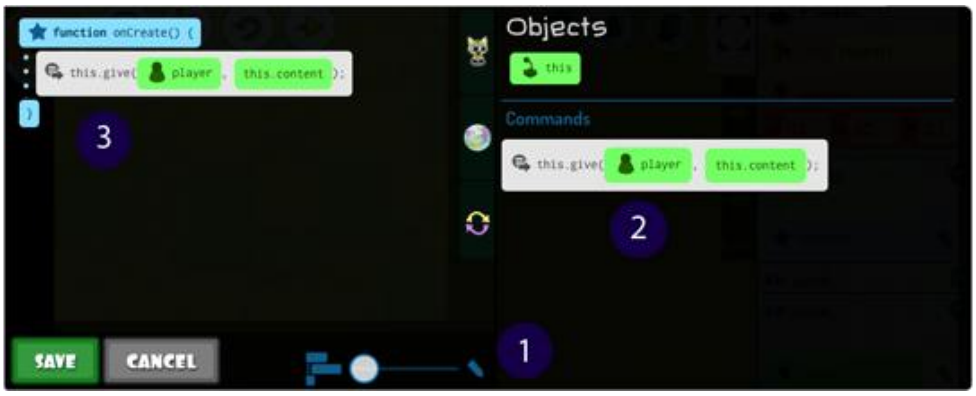


Figure 7.6: Code Kingdoms World Design Editor

The Code Kingdoms environment is event-based, and can be programmed using nested visual blocks (Figure 7.7). In addition to visual blocks, the environment allows users to manipulate text code by providing a slider that converts the blocks into text in four stages: annotated blocks, simple blocks, drag and drop text, and simple text (Figure 7.6)

The founders of Code Kingdoms regularly carry out workshops in schools in order to gain feedback, discover bugs in the software, and encourage students to use the system at home. Each workshop is 1hr 30m long, and covers the creation of a world, and how to program buttons and characters at increasing levels of complexity (from assignment, to function calls, and finally iteration).



function onCreate() {
this.give(player , this.content);
}
3
Objects
this
Commands
this.give(player , this.content);
2
SAVE
CANCEL
1

Figure 7.7: Code Kingdoms In-browser Development Environment

Top: ❶ Block to text switcher ❷ Command Area ❸ Code Area

7.5 Investigation Design

In this investigation, a between-subjects study was conducted to examine the validity of four candidate assessment tools for measuring Notational Expertise in a classroom environment. The assessment tools – Code Writing, MCQ, Adapted Parsons Problems, and Debugging, were given to classes of students during a Code Kingdoms workshop.

7.5.1 Participants

To compare and investigate the suitability of each assessment type, four different classes of students were recruited from two schools. Each class was given a different assessment. To increase internal validity, each class was selected to ensure that all classes had a similar range of ability.

Table 7.1: Assessed Syntax Features

Question	Type	Expected Answer
0 (Example)	Declaration	`var x`
1	Assignment	`myVariable = 2;`
2	Member Access	`GlitchB.age = 10;`
3	Method call	`helperA.say(“Hello!”);`
4	Method call	`GlitchC.stop();`
5	Add Operator	`myVariable = 2 + 2;`
6	Multiply Operator	`myVariable = 2 * 2;`
7	If statement	`if(player.alive){` `GlitchC.exlaim();` `}`
8	While loop	`while(GlitchA.alive){` `this.exlaim();` `}`
9	While loop	`while(True) {` `this.walk(NORTH);` `}`

7.5.2 Apparatus

The investigation took place in classrooms in two schools in Cambridgeshire and Berkshire. In the Cambridgeshire School students used laptops on their desks, whereas in the Berkshire school students used PCs in a computing suite. The classroom arrangement and teaching process was designed to be as similar as possible to an ordinary computing lesson, in order to increase the ecological validity of the investigation.

At the end of the CK workshop, participants were given questionnaires and assessments to complete on paper, during which time PC and laptop screens were turned off.

7.5.3 Procedure

Participants were given a short introduction to the Code Kingdoms system followed by an 85-minute workshop in which students used example CK levels to understand the type of levels that could be built, and started building their own levels with the goal of adding functionality to objects and characters using the integrated JavaScript editor (Figure 7.7). At the end of the workshop, students were given half an hour to answer a background questionnaire, 9 questions (Table 7.1) from the assessment given to their group, and a post-assessment questionnaire.

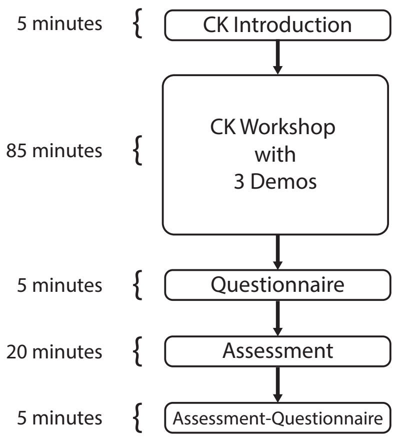


Figure 7.8: Investigation Procedure

7.5.4 Data Collection

The questionnaire, assessment, and assessment questionnaire were all delivered to students on paper. Each item was scanned, transcribed and coded for analysis. Instrumentation data from Code Kingdoms was collected via integrated web analytics in order to ensure that each student had completed the expected tasks during the workshop.

7.5.5 Measurements

To evaluate and compare candidate assessments for measuring Notational Expertise, several techniques were adopted from related literature on CS1 evaluation, presented in section 7.2. In particular, Denny, Luxton-Reilly, and Simon's work on evaluating new exam questions, in which the authors compared Parsons Problems to Code Writing questions using correlation and individual question comparison (Denny et al., 2008).

Marking Rubric

A consistent marking rubric was used for both Parsons Problems and code writing assessments; each syntax element required for the answer was worth one mark for inclusion, and one mark for being in the correct position. Debugging answers required 1-2 additions, deletions or modifications, which were given a mark each. MCQ answers were given one mark for being correct, and no marks for being incorrect. Results from each assessment were normalised to a percentage for comparison.

The post-assessment questionnaire asked participants to rate the assessment using Likert questions for difficulty, enjoyment, and difficulty compared to other assessments. Participants were asked to describe the best parts, the worst parts, improvements they would make, and any extra comments.

7.6 Investigation Results

80 participants – 42 boys and 38 girls from ages 10 to 11, were recruited from two primary schools in Cambridgeshire and Berkshire to participate in Code Kingdoms workshops. Participants had a range of academic ability.

Table 7.2: Participant Grouping

School	Group	Participants	Gender Split
Cambridgeshire	Code Writing	22	10M / 12F
Cambridgeshire	Debugging	25	14M / 11F
Berkshire	MCQ	17	9M / 8F
Berkshire	Parsons	16	9M / 7F

Of the 80 participants recruited, 67 participants had some animation experience, 35 had some programming experience, and 15 had some webpage development experience.

7.6.1 Formative Assessment Comparison

Results show that the MCQ assessments measured the mean highest mark (M=55%) and the largest range (SD=22.91%), followed by Parsons (M=48%, SD=9%), Debugging (M=33%, SD=12%) and Code writing (M=9%, SD=16%). A Shapiro-Wilk test indicated that test data was not normally distributed $W = 0.9678, p = 0.041$. A Kruskal-Wallis rank sum test showed that there was a significant difference between test averages by assessment type $F(3,76) = 22.61, p = 1.45\, e^{-10}$.

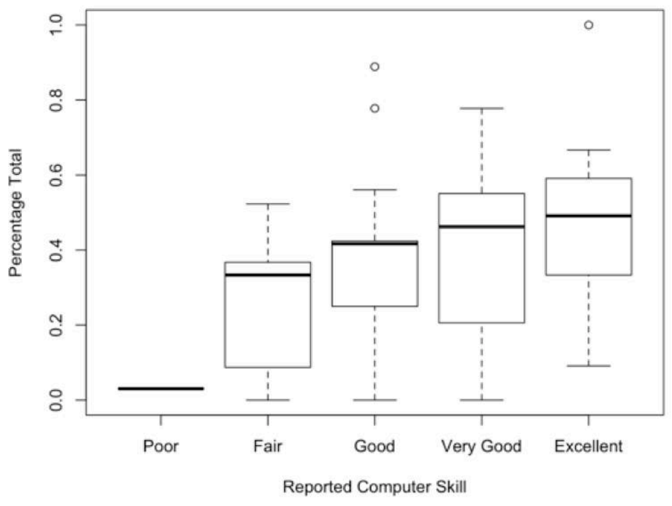


Figure 7.9: Reported Computer Skill

Figure 7.9 illustrates student test score distributions (over all assessments) separated by their reported computing skill. A Kruskal-Wallis test found that the average student test mark varied relative to their reported computer skill in the background questionnaire, suggesting that students had accurate levels of confidence before completing the assessment $\chi^2(4, N = 80) = 12.99, p = .0113$.

Question Difficulty

Results show that Question A, E and F, which addressed assignment, the addition operator, and the multiply operator respectively, were the highest scoring questions in Parsons and code writing assessments (Table 7.3). The lowest scoring questions in all assessments, G and H, addressed multiline if and while loops.

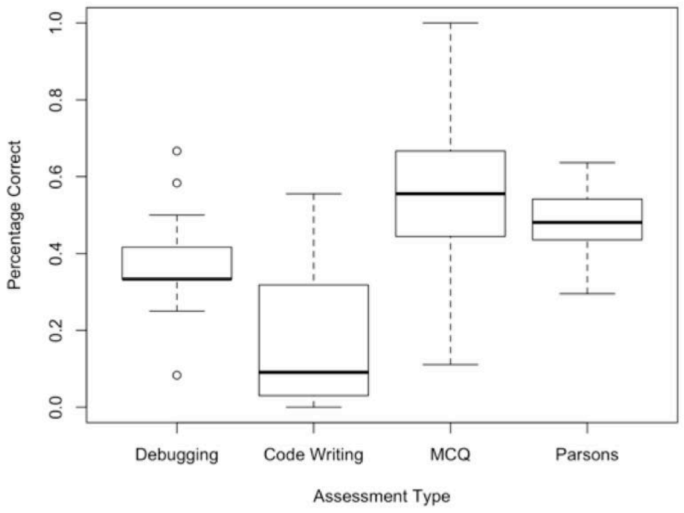
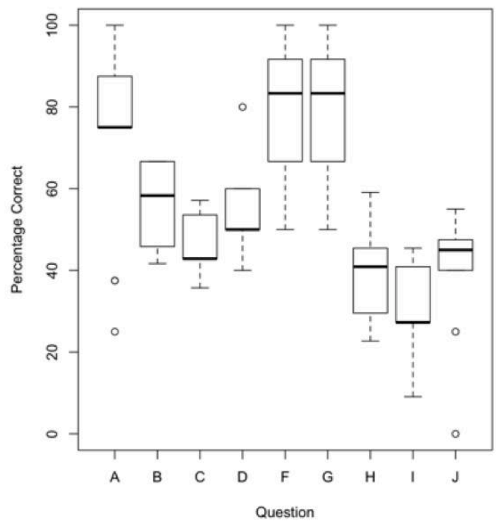


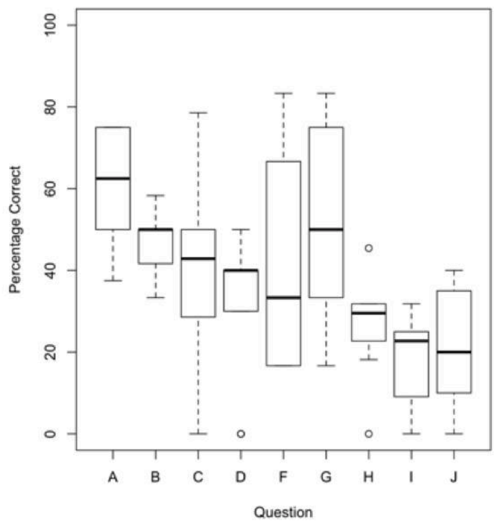
Figure 7.10: Assessment Results Comparison

Table 7.3: Normalised Mean Assessment Results by Question

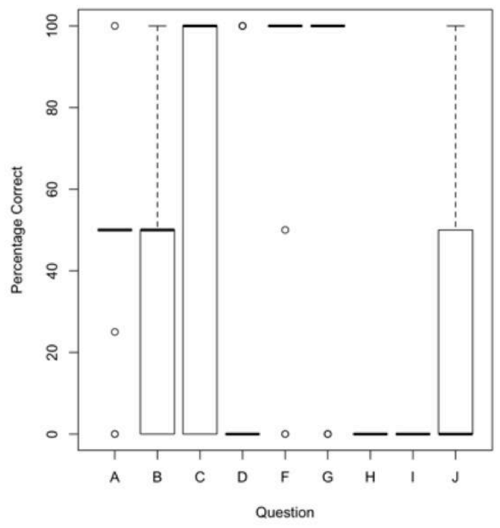
	A	B	C	D	E	F	G	H	I
Debugging	47	36	56.5	23.8	89.6	79.2	0	0	26.2
Writing	61.7	46.4	41.6	31.1	45.4	50	26.7	17.5	21.4
Parsons	74.2	56.7	45.9	53.1	80.7	80.7	37.9	30.5	41.2
MCQ	58.8	58.8	64.7	47.1	82.4	76.5	41.2	29.4	40



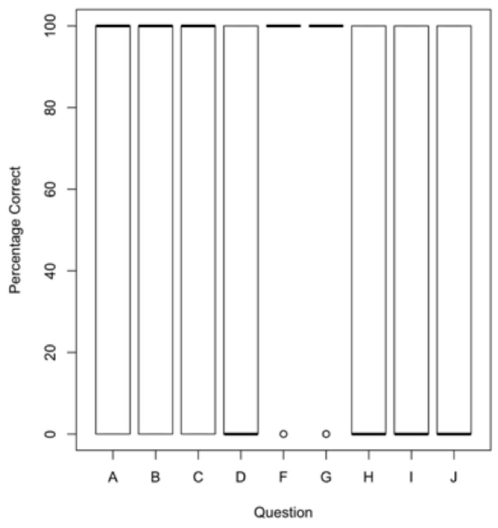
(A) Parsons Mark Distribution



(B) Code Writing Mark Distribution



(C) Debugging Mark Distribution



(D) MCQ Mark Distribution

Figure 7.11: Mark Distributions Per Question

Code Writing

There was a low response rate to code writing questions, with participants responding to less than half of the total number of questions given in the assessments (83 of 198 questions, 41.92%). Six participants did not answer any questions. In the assessment-questionnaire, the majority of participants reported that they found this assessment difficult or very difficult (18/21), stated they disliked or hated it (15/21) and stated that it was harder or a lot harder compared to other assessments they had done (16/21). Students noted that the worst things about the assessment include "everything was hard", "changing writing into code"

Figure 7.11-A and Figure 7.11-B illustrate the similarity between the distribution of marks in code writing questions and Parsons Problems. A Pearson's product-moment correlation shows that there may be a positive correlation between average results scored on each test

($r = 0.32, p = 0.2344, R^2 = 0.099$). The same test shows little correlation between code writing and debugging assessments ($r = -0.02, p = 0.9312, R^2 = 4 . 10^{-4}$), and code writing and MCQ assessments ($r = 0.2, p = 0.4451, R^2 = 0.04$). A further correlation performed between the frequency distributions of code writing and Adapted Parsons Problems found that they were positively correlated ($r = 0.31, p = 0.00032, R^2 = 0.064$).

Debugging Assessment

There was a high response rate to debugging questions, with the majority of participants giving a response (201 of 225 questions, 89.33%). The mean score was 37% with a standard deviation of 12.04%. Almost half of participants reported that they found this assessment difficult or very difficult (10/24), with the same number reporting to dislike or hate the assessment type, and half of participants reporting it was harder or a lot harder than other assessments they had done.

MCQ Assessment

There was a 100% response rate to MCQs. The mean score was 55.56% with a standard deviation of 22.91%. More than half of the participants (9/17) stated they found the MCQs assessment difficult. However, the majority (15/17) stated that they were indifferent, liked or loved the assessment. Over half stated that it was harder than other assessments they had done (11/17).

Adapted Parsons Assessment

There was a high response rate to Parsons Problems, with the majority of participants giving a response (136 of 144 questions, 94.44%). The mean score was 48.15%, with a standard deviation of 9.04%. Half or participants stated that they found the assessment difficult or very difficult (8/16). Less than half reporting feeling negative about the assessment (6/16), and more than half (10/16) reported that it was harder or a lot harder than other assessments they had done.

7.6.2 Gender Differences

Males rated their computer skill more highly than females, with 12 males rating themselves as "excellent" compared to 2 females overall. Males were also more likely to have programmed; 50% of males responded they had programmed (21/42) compared to 37% of females (14/38).

An unpaired t-test showed that there was a significant difference in test-average between genders on MCQs assessments $t(2.80), p = 0.014$, with males (n=9) scoring 68% on average, and females (n=8) scoring 41.7%. There was no significant difference in any other test. Males for this group rated their computer skill more highly than females.

7.7 Discussion

Similarity to Code Writing

The aim of this investigation was to identify the most appropriate assessment type for measuring Notational Expertise in students at KS2 and KS3 in future studies. Code writing tasks are still used in the majority of programming assessments (Simon et al., 2012), but are known to be difficult to mark (Denny et al., 2008), and require complex cognitive processes (Krathwohl, 2002). Results in this investigation indicate that Code Writing questions provide a reasonable distribution of marks over each of the 9 questions. However, student response-rate and questionnaire feedback indicated that code writing may have been intimidating, and could have had a negative bearing on student confidence and therefore self-efficacy.

Results indicated that Adapted Parsons Problems had the highest level of correlation with code writing questions, and had a significantly correlated frequency distribution, in addition to achieving more than double the response rate and more desirable student feedback in the post-assessment questionnaire.

The process of designing MCQ and Debugging questions suggested that they were likely to cause similar issues to those encountered in the previous chapter. Although there were no known observations of students using similar strategies in this investigation, results from both the debugging assessment and MCQ assessment did not correlate with code writing results, suggesting the assessments were either susceptible to inferences or measuring a different skill.

Participation and Motivation

Observations of each assessment, participation levels, and results from post-assessment questionnaires suggest that students would be highly demotivated by code writing assessments, which could negatively affect any further study and more concerning, the students' interest in programming in the future. Participation for non-code writing assessment types was high (> 90%), with questionnaire feedback positive in the case of MCQs, and middling in the case of Debugging and Adapted Parsons Problems.

MCQs

The wide distribution of marks for the MCQ assessment would suggest it is likely to be a useful tool for differentiating students as part of a summative assessment. However, unexpectedly, the significant difference between male and female performance in MCQ results suggest that there may be a gender bias, possibly due to high confidence in males.

Question Difficulty

Consistencies between assessments show that some questions, such as those relating to while loops, were more difficult, while others, relating to operators that are familiar from mathematics, were easier. While teachers may prefer a mixture of difficulty levels to elicit a wide distribution of marks, it is preferable in studies of Notational Expertise, to measure understanding against a fixed set of representational features so that the effectiveness of DrawBridge can be evaluated.

Threats to Validity

Researchers carrying out similar work to compare assessments for measuring understanding in programming have used integrated assessments to evaluate new questions (Denny et al., 2008; Murphy et al., 2012). In these assessments, new questions are posed alongside conventional code writing questions to allow researchers to carry out within-subjects analysis. This investigation used a between-subjects design to avoid student demotivation resulting from confusion and high cognitive load. To increase the validity of comparisons made between groups, participants were recruited from schools with similar academic performance according to Ofsted (Cambridgeshire – 94% level 4 reading, 93% level 4 writing, 86% level 4 maths; Berkshire – 94% level 4 reading 93% level 4 writing, 93% level 4 maths).

To ensure consistency between assessments, and to avoid confusion by using jargon students may not have been familiar with after such a short time programming, each of the 9 questions in each assessment used the same wording, which contained as little jargon as possible. This resulted in some questions that may have been too vague. For example, question C was worded "Make GlitchC say 'Hello!'", rather than "Call the method "say" of object GlitchC passing the parameter 'Hello!'". This trade-off may have been a contributing factor to the low response rates achieved in the Code Writing assessment. However, post-assessment questionnaire feedback, and observation during the assessment indicate that students found creating code difficult and intimidating.

The investigation was conducted during four Code Kingdoms workshops given in two different schools, for two reasons: (1) all participants had a consistent workshop before they completed their assessments, and (2) Code Kingdoms is a fully functional, commercial novice programming system, which has no affiliation with DrawBridge, increasing the external validity of the results.

7.8 Chapter Summary

This chapter presented an investigation to evaluate and compare four candidate assessments for measuring Notational Expertise with students in KS2 and KS3. The motivation for this investigation was to address concerns raised from the study carried out in the previous chapter, and by Gross and Powers (Gross & Powers, 2005) who claimed that novice programming environment assessments are too opportunity driven, and difficult to repeat.

The research question asked which type of assessment would be most appropriate for measuring Notational Expertise. The investigation found that conventional code writing questions elicited poor response rates, and highly negative responses from students completing the assessment. Results from Adapted Parsons Problems were found to correlate significantly with code writing questions, while maintaining high student response rates and satisfactory post-assessment questionnaire feedback. MCQ and debugging assessments had no correlation with code writing, and were found to suffer from design issues such as those experienced in the assessments given in the previous chapter. Adapted Parsons Problems will therefore be used in future studies to measure NE.

Chapter 8 Motivation Intervention to enhance Notational Expertise

8.1 Introduction

The first study to make use of the DrawBridge MER system, presented in Chapter 6, found that students who had access to all representation types spent the majority of their time using low-abstraction representations, which were designed to provide motivation and scaffolding, but actually resulted in distraction. Based on those findings, this chapter describes the design, implementation and evaluation of new motivation intervention features in DrawBridge, which are intended to encourage students to make the transition towards symbolic notations and thereby improve acquisition of Notational Expertise (NE).

As in Chapter 6, to evaluate new features it was preferable for DrawBridge to be studied in a classroom using existing classes of students and an existing computing lesson format. To support this, it was necessary to employ an experimental design that accounted for non-randomised treatment and control groups, and supported the use of qualitative data such as observations to assist in the discussion of quantitative findings.

This chapter therefore presents two quasi-experiments, which compare classes of students using new motivation intervention features in DrawBridge with students using a standard version. This method enables the identification of problems and interactions that arise during the use of such a system in an authentic context. The investigation and discussion of the studies reported here will be used to validate and refine design goals of the system, and ultimately provide guidance for designers of new MER systems. Each study addresses three research questions established in Chapter 6:

RQ5: To what extent can motivation intervention features increase the use of symbolic representations and therefore improve acquisition of notational expertise?

The first research question emerged from a study investigating strategies for representation transition in DrawBridge, presented in Chapter 6. Observations during that study, and subsequent analysis of instrumentation data, revealed that students were spending the majority of their time viewing low-abstraction, or "concrete" representations. Feedback from students confirmed this result, and showed that concrete representations in DrawBridge provided a rich source of motivation. This result lead to the question of whether students could be encouraged to make the transition to symbolic representations more quickly, and therefore improve their ability to acquire NE.

To investigate this question, DrawBridge was extended to include motivation intervention features, which were developed by applying findings from research literature in Computing Education, Persuasive Computing and Gamification. Increased understanding of the effectiveness of motivation intervention features may be important in the development of future MER systems for educational programming, and in improving use of existing systems which contain sources of motivation that may prove to be distracting.

RQ6: To what extent can student year group affect the acquisition of NE in DrawBridge?

The second research question was concerned with the year group in which students use MER educational programming environments, and how beneficial they might be to students from lower year groups. This question emerged from three different sources: Piaget's model of cognitive development, the quasi-experiment reported in Chapter 6, and changes to the UK national computing curriculum.

Piaget's stages of cognitive development suggest that children achieve the final stage of development (Formal Operational) during adolescence, and can consequently reason about abstract operations through the use of symbols. Young children who may not have developed the cognitive function required to carry out such reasoning, and therefore use the system, may suffer from reduced self-efficacy due to a high level of *Hard Mental Operations* $^{(CD)}$, coupled with the increased cognitive load inherent in MER environments (Ainsworth, 1997).

The first study to make use of DrawBridge, reported in Chapter 6, found that assessment questions that required translation elicited low levels of response from Year 7 students. An understanding of how different year groups respond to DrawBridge, and assessments to measure Notational Expertise, would help to determine whether a system and accompanying assessments are appropriate, and whether changes to either are required so that they can be used with Year 7 students more effectively.

New changes implemented in the UK national computing curriculum advise that students at Key Stage 1 (Year 1 - 2) should create and debug simple programs, students at Key Stage 2

(Year 3 - 6) should design, write and debug programs, and students at Key Stage 3 (Year 7 - 9) should be able to use two or more programming languages, at least one of which is text-based (DfE, 2013). Such recommendations raise the question of whether students in lower year groups can benefit from using MER programming environments, and if so, whether the benefits they receive change dependent on year group.

RQ7: To what extent do motivation intervention features affect acquisition of NE in students of different year groups using DrawBridge?

The third research question is concerned with the interaction of the first two research questions. The addition of motivation intervention features may cause increased *Hard Mental Operations* $^{(CD)}$ when using the system, due to added complexity to each representation, and the relationship between representations. If students in lower year groups receive less benefit from using the system, then the addition of motivation intervention features may be problematic, reducing or removing any benefit.

Based on the research questions described above, an extension to DrawBridge was designed and implemented to encourage users to explore symbolic representations after a fixed number of interactions with concrete representations.

The chapter is structured as follows. Section 8.2, reviews the literature in computer science education assessment, Design with Intent and Gamification. Section 8.3 describes changes to DrawBridge that facilitate investigation of the three research questions. Section 8.4 describes the pilot study with Year 7 students with corresponding required improvements and changes to procedure. Section 8.5 presents the design for Study 1, which consisted of a controlled classroom study to investigate effectiveness of new motivation intervention features using an integrated assessment. Section 8.6 describes the results of the study, with discussion in section 8.7. Section 8.8 presents the design of a second, follow up study, which further investigates research questions presented above. Section 8.9 presents the results of the follow-up study, with a discussion in section 8.10. Finally, section 8.12 reports the combined findings of both studies in the context of the three research questions.

8.2 Related Work

Developing persuasive technology is not a new goal for HCI researchers. In his well-known book The Design of Everyday Things, Norman (Norman, 1988) describes objects by their *affordances*: perceived and actual properties of an object, *constraints*: limitations of an object, and *mappings*: the set of possible operations the user can perform on the object. Norman argues that the designer can shape the object interaction by finding the correct mixture of these elements.

8.2.1 Persuasive Interventions and Gamification

The Design with Intent framework (DwI) extends Norman's work to form a cross-disciplinary approach that includes persuasive ideas from Manufacturing, Philosophy and Architecture (Lockton, Harrison, & Stanton, 2008). The framework offers practical design patterns that can be used to influence interactions between the user and the system, such as task lock-in/out to encourage the user to follow a specified path or process, and giving the users functionality when environmental criteria are satisfied (Lockton, 2015). These patterns were used in the development of motivation intervention features to restrict the use of low-abstraction concrete representations using task lock-out, which stopped users from using certain representations too much, and path guidance, where users were encouraged to increase their use of symbolic representations.

Motivation for changes in interaction can also be achieved through communication with the user. In Computing Education research, Pedagogical Agents – visually represented, computer generated characters that act in a pedagogical role as an instructor, or learning companion, (Haake, 2009), such as Gidget (Lee & Ko, 2012), have been used to successfully engage students in working towards education goals. Although some results using pedagogical agents have been encouraging, evidence for their effectiveness has been mixed, and is difficult to generalise due to most evaluations consisting of a single study (Gulz, 2004).

Huotari defines "Gamification" as "*a process of enhancing a service with affordances of game-like experiences in order to support users' overall value creation*" (Huotari, 2012). So-called "gameful" experiences, which intend to influence behavioural outcomes, include the use of achievements, points, leader boards, badges, levels, and challenges. Empirical studies have found that gamification can be used to achieve positive results (Hamari, Koivisto, & Sarsa, 2014), which suggests that such features may be used to encourage students to make the transition to symbolic representations in MER systems. However, care must be taken to avoid mistakes made by systems such as Khan Academy, which have been criticised for failing to relate gamification features to underlying activities, and potentially creating hollow user experiences (Morrison & DiSalvo, 2014).

8.2.2 Attention Investment

Blackwell argues that the decision of whether to learn to use abstract programming notation or rely on Direct Manipulation to complete a task relies on an implicit cost/benefit analysis (Blackwell, 2002). This analysis can be modelled using attention units, based on *cost* of the work, *investment* towards it, *pay-off* to reduce cost in future, and *risk* that the investment will not pay off. I argue that in educational programming environments there is a similar cost/benefit analysis, in which students consider using abstract symbolic notations when compared with Direct Manipulation. However, the pay-off is not only the reduced cost as a result of automation in the future, but also the intrinsic value in learning to program, and access to functionality using abstract notation that cannot be accessed with Direct Manipulation. The addition of extra incentives is likely to persuade students that the benefits of using abstract notations outweigh the costs of learning to use them.

8.2.3 Instrumental Conditioning

In Psychology, the process of learning in response to positive and negative outcomes resulting from ones behaviour is known as Instrumental Conditioning. As in gamification, outcomes can consist of punishment for incorrect behaviour, and reinforcement for correct behaviour. Thorndike's Law of Effect states that behaviour that is accompanied or closely followed by satisfaction, will be firmly connected to the situation in which it occurred, so that when it recurs, the behaviour is more likely to be repeated, changing behaviour over time (Lieberman, 1999). Although examples of conditioning are typically described via empirical experiments of animal behaviour, the process of reinforcing behaviour may be relevant to encouraging students to make the transition between representations.

The related work described above was used to develop novel motivation intervention features in DrawBridge to encourage students who might otherwise spend all their time using low-abstraction, concrete representations, to make the transition to high abstraction symbolic notations.

8.3 System Design

The first version of DrawBridge (herein referred to as v1) was used for early studies investigating the effects of manipulating the order of representations (see Chapter 6). In order to examine new research questions arising from those studies, and a further study comparing assessments (see Chapter 7), a second iteration of development was required. The new iteration of DrawBridge, (v2), would include motivation intervention features, integrated assessment features and general improvements, which were gathered from participant reaction, observations, and questionnaire feedback, presented in Chapter 6.

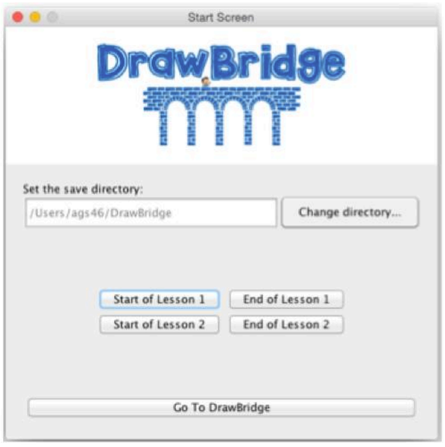
Table 8.1: Table of Features for DrawBridge 2

#	Feature	Task Size
1	Add Motivation Intervention capability	Large
2	Integrate Adapted Parsons Problems	Large
3	Improve style of user code	Medium
4	Improve syntax annotation	Medium
5	Add ability to load/save DrawBridge sessions	Medium
6	Add customisable backgrounds	Medium
7	Change milliseconds to seconds	Small
8	Trigger advice popups when slider hit limits	Small

8.3.1 Assessment Integration

To separate Adapted Parsons Problems from other parts of DrawBridge, a new window was created that could only be accessed when the main DrawBridge window had been closed. The separation served two purposes: to prevent students from becoming distracted during their assessment, and to reduce the chance of students viewing scaffolding code that could help them when answering assessment questions.

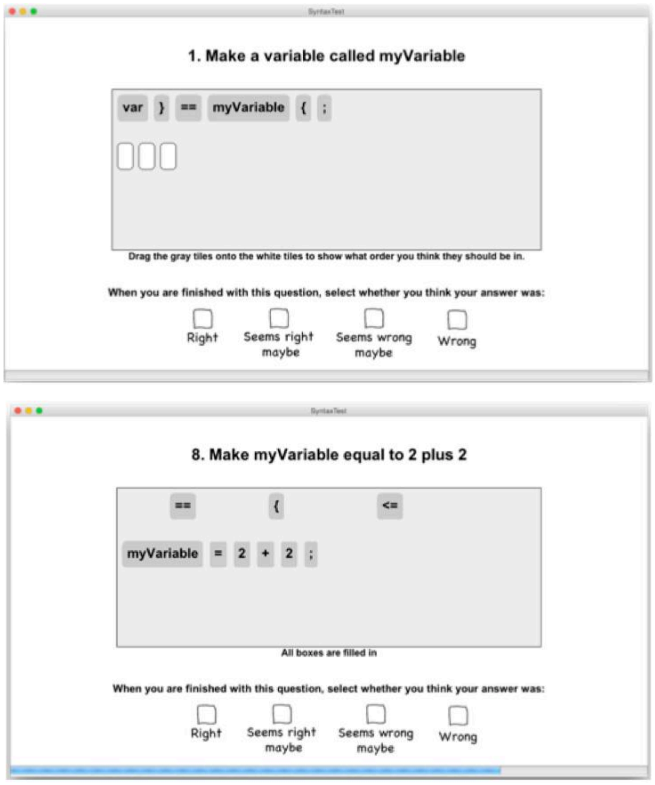
An example of the assessment window, shown in Figure 8.2, illustrates that in order to answer each question, the user must choose a tile from several possible syntax elements (correct elements and distractors) and drag it to the appropriate position in the answer. When the user begins to answer, a feedback text field below the answer box displays the number of empty boxes left. When the user has completed their answer, they must select from one of four confidence levels, inspired by Beckwith et al's evaluation checkboxes (Beckwith et al., 2005): *Right*, *Seems right maybe*, *Seems wrong maybe* and *Wrong* before proceeding to the next question. The assessment progress is displayed in a progress bar that appears at the bottom of the screen. A start screen, shown in Figure 8.1, was added to make assessment features accessible in DrawBridge, and to allow users to locally save and load DrawBridge sessions.



Set the save directory:

/Users/ags46/DrawBridge

Figure 8.1: DrawBridge Start Screen



1. Make a variable called myVariable

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

8. Make myVariable equal to 2 plus 2

All boxes are filled in

When you are finished with this question, select whether you think your answer was:

Figure 8.2: DrawBridge Integrated Assessment

(**Top**: Unanswered question **Bottom**: Answered question without confidence rating.)

8.3.2 Coins as Motivation Intervention Features

DrawBridge was extended to include novel features referred to as "motivation intervention" features. These features were designed to gently encourage participants towards increasingly abstract representations and thereby circumvent issues encountered during studies using DrawBridge V1 (see Chapter 6), in which participants spent a large proportion of their time using more concrete representations.

To limit the amount of interaction with non-abstract representations, each interactive panel in DrawBridge was extended to show an allocation of "coins". On opening DrawBridge, the user receives a pre-determined number of coins for each panel, which is set depending on a fixed pre-defined weight given to each panel (see Equation 2). Users "spend" one coin for every interaction they carry out on a given panel (see Equation 3). If a coin is spent, it is redistributed to other panels according to predefined weights (see Equation 4).

$$Coins_i = \omega_i \, * \, 30$$ **Equation 2**

$$Coins_i = \; Coins_i - 1$$ **Equation 3**

$$Coins_j = \; Coins_j + \frac{\omega_j}{\sum_{p=1}^{k} \omega_p}$$ **Equation 4**

Both Direct Manipulation panels are given a weighting of 1, while the Visual Block Panel are given a weighting of 3, and the Text Panel is given a weighting of 6. The interaction required to spend a coin is shown in Table 8.2.

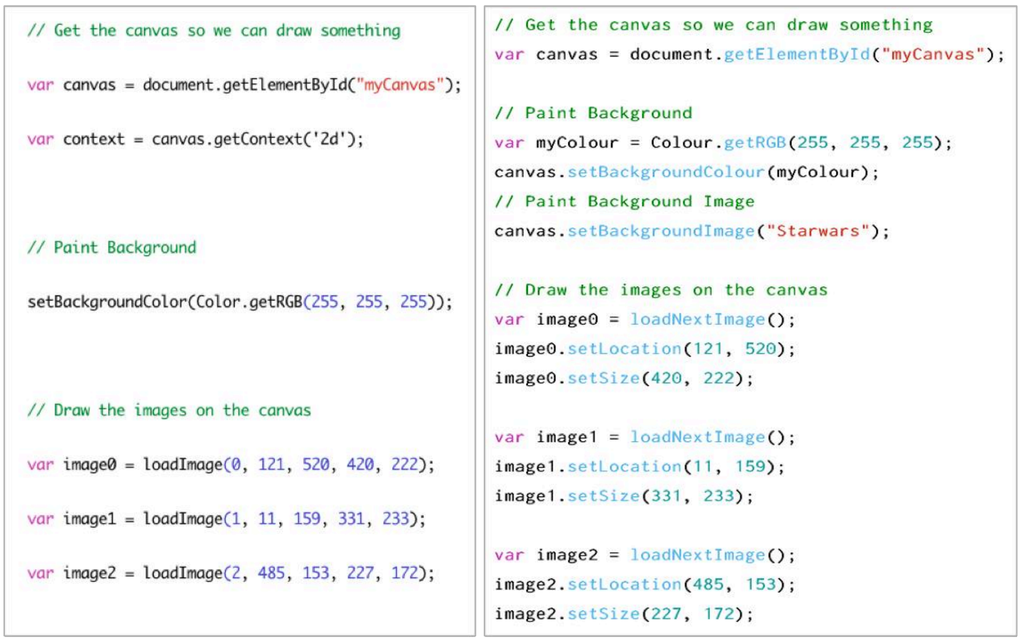
As a working example, consider the user opening DrawBridge and starting with {30, 30, 90, 180} coins on the Direct Manipulation, Abstract Direct Manipulation, Visual Block and Text Panel respectively. If the user dragged and dropped a new block to the Visual Block Panel, they would use up one token, resulting in {30, 30, 89, 180}. The coin would then be redistributed to the other panels, resulting in {30.125, 30.125, 89 180.75}. The number of coins available for each panel is displayed in its title bar. As the number of coins drops below 10, the number flashes for every subsequent interaction. When the panel has no coins left, it becomes disabled, flashing "0 coins left" in the title bar when the user attempts to interact with it.

Table 8.2: Panel Coin Relationship

Panel	**Coin Allocation**	**Interaction per Coin**
Paper	-	-
Direct Manipulation	30	Every click
Abstract Direct Manipulation	30	Every click
Visual Block	90	Modification of blocks, dialling values
Text Code	180	Character entry
Web Preview	-	-

8.3.3 Technical Improvements

A number of technical improvements were required to address bugs or usability issues in the system. For example, the boilerplate code for loading images in DrawBridge v1 used a function “`loadImage`”, which took an index as one of its parameters. Although students were not required to understand why the index parameter was needed, it may have been a point of confusion that could be avoided. The function was removed, and replaced by “`loadNextImage`” which takes no parameters and returns an image object or Nil. The other parameters taken in “`loadImage`”, such as position and size, were separated out into the image object functions “`setSize`” and “`SetLocation`”. These functions take two parameters each, and make the invisible semantics of the “`loadImage`” function they replace more obvious to novice users.



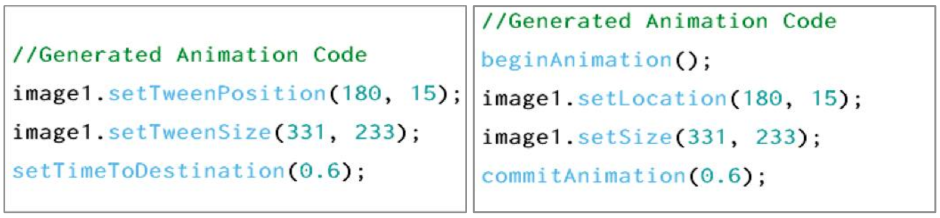
```
// Get the canvas so we can draw something

var canvas = document.getElementById("myCanvas");

var context = canvas.getContext('2d');



// Paint Background

setBackgroundColor(Color.getRGB(255, 255, 255));



// Draw the images on the canvas

var image0 = loadImage(0, 121, 520, 420, 222);

var image1 = loadImage(1, 11, 159, 331, 233);

var image2 = loadImage(2, 485, 153, 227, 172);
```

```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);
// Paint Background Image
canvas.setBackgroundImage("Starwars");

// Draw the images on the canvas
var image0 = loadNextImage();
image0.setLocation(121, 520);
image0.setSize(420, 222);

var image1 = loadNextImage();
image1.setLocation(11, 159);
image1.setSize(331, 233);

var image2 = loadNextImage();
image2.setLocation(485, 153);
image2.setSize(227, 172);
```

Figure 8.3: Boilerplate Code Modifications in DrawBridge v2

Left: DrawBridge v1, **Right**: DrawBridge v2

In addition to changes in boilerplate code, animation code generated using Programming by Demonstration (PbD) on the Abstract Direct Manipulation panel was modified to take a transactional approach, in which the code for each animation step is contained between two functions, "beginAnimation" and "commitAnimation". This approach makes each step of the animation explicit – improving on the old implicit code, which used image methods to set the animation size/position and then a function called "setTimeToDestination" to trigger each animation (see Figure 8.4).



```
//Generated Animation Code
image1.setTweenPosition(180, 15);
image1.setTweenSize(331, 233);
setTimeToDestination(0.6);
```

```
//Generated Animation Code
beginAnimation();
image1.setLocation(180, 15);
image1.setSize(331, 233);
commitAnimation(0.6);
```

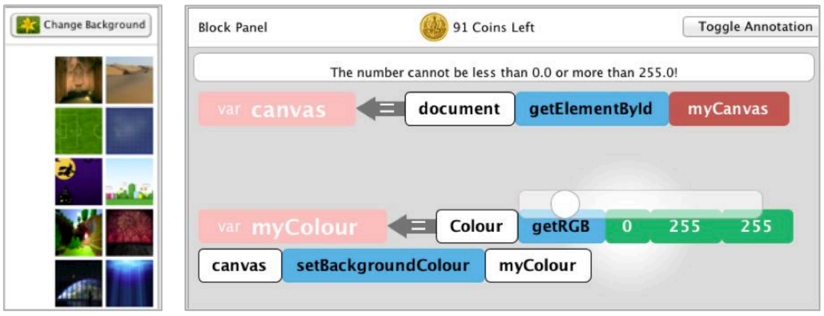
Figure 8.4: Animation code changes in DrawBridge

Left: Version 1 animation code, **Right**: Version 2 animation code.

In DrawBridge v1, blocks in the Visual Block panel were annotated with a subset of JavaScript syntax. For example, the dot between an object and its method call was drawn implicitly if a function block was placed next to an identifier. In version 2, the translation system was extended to include a block parser, which generates annotation based on the relative position of each block. The annotation includes all of the JavaScript syntax supported in DrawBridge, including brackets, curly brackets, semi-colons, quotes and parameter commas. Users were given the option to turn off annotation using a button placed in the title bar of the panel in case they found the annotation distracting or confusing.

The smaller features mentioned in feedback questionnaires were designed and implemented to be as unobtrusive as possible. DrawBridge was modified so that all assets were saved every time the code was tested. When opened, DrawBridge checks the given directory for user code and images, and loads them if available. Users also commonly requested the ability to select a background. Ten backgrounds were supplied via a dropdown background selector placed in the title bar of the Direct Manipulation Panel. The feature required the addition of a corresponding method called "`setBackgroundImage`", which takes a string as a parameter.

The think-aloud study presented in Chapter 6, suggested that students did not understand some boilerplate code parameters, such as the parameter to specify the duration of an animation, which was specified in milliseconds. To address this, animation durations in DrawBridge v2 were modified to use seconds. Additionally, students found it difficult to understand that the background colour can be specified using three numbers (RGB - Red, Green and Blue) between 0 and 255. To make these limits clearer, a warning popup was added to the Visual Block Panel to inform the user that they could not go beyond the limits in either direction.



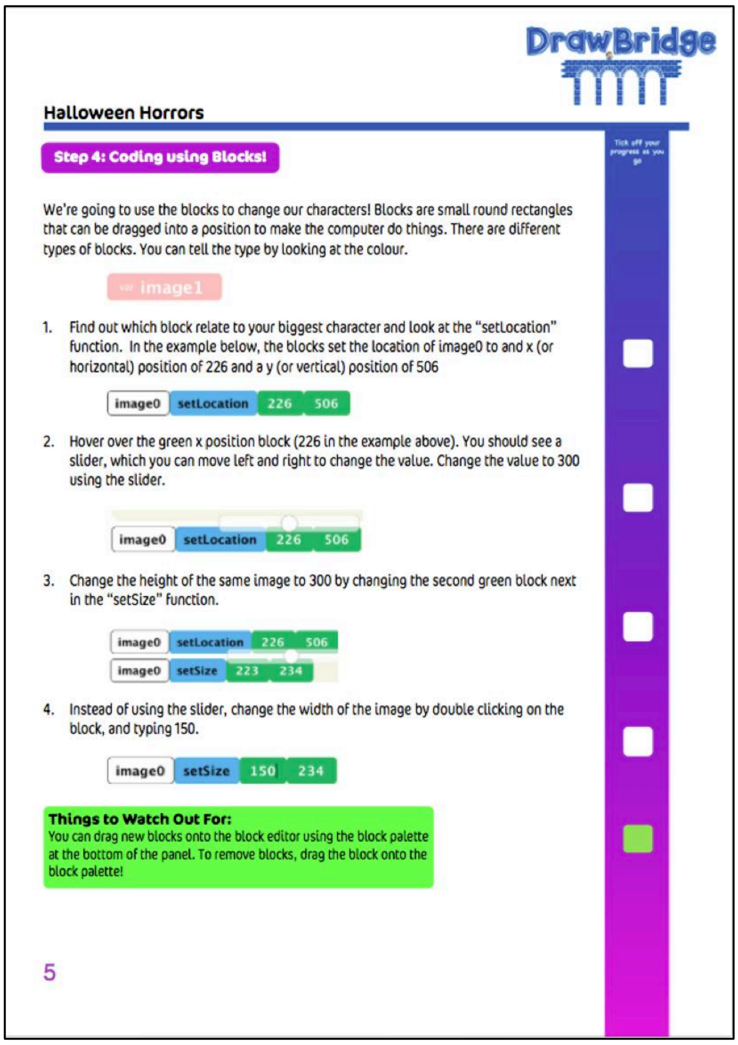
The number cannot be less than 0.0 or more than 255.0!

```
var canvas = document getElementById myCanvas
var myColour = Colour getRGB 0 255 255
canvas setBackgroundColour myColour
```

Figure 8.5: Background Selector (Left) and Limits Warning Popup (Right)

8.3.4 Workbook Updates

In addition to improvements and new features in the system, the accompanying DrawBridge workbook was improved and split into two parts: (1) a follow-me tutorial giving students examples and directions on how to use DrawBridge, and (2) descriptions of underlying concepts such as variables, functions and objects. Figure 8.6 shows examples of pages from both workbooks, which give simple step-by-step instructions, examples of code, and things to watch out for.


DrawBridge

Halloween Horrors

Tick off your progress as you go

Step 4: Coding using Blocks!

We're going to use the blocks to change our characters! Blocks are small round rectangles that can be dragged into a position to make the computer do things. There are different types of blocks. You can tell the type by looking at the colour.

1. Find out which block relate to your biggest character and look at the "setLocation" function. In the example below, the blocks set the location of image0 to and x (or horizontal) position of 226 and a y (or vertical) position of 506

2. Hover over the green x position block (226 in the example above). You should see a slider, which you can move left and right to change the value. Change the value to 300 using the slider.

3. Change the height of the same image to 300 by changing the second green block next in the "setSize" function.

4. Instead of using the slider, change the width of the image by double clicking on the block, and typing 150.

Things to Watch Out For:
You can drag new blocks onto the block editor using the block palette at the bottom of the panel. To remove blocks, drag the block onto the block palette!

5

Figure 8.6: Example of DrawBridge Worksheet

(Page 5 from Workbook 1. For more, see Appendix D)

8.4 Pilot Study

A pilot study was carried out in order to validate the study procedure and identify any technical limitations using DrawBridge v2 within a school's computing infrastructure. The pilot study was conducted with six male Year 7 students at a weekly after-school IT club in an independent school in West Yorkshire. In addition to identifying technical issues and verifying procedure, the pilot study enabled validation of new questionnaires, and produced data that could be used to evaluate the method of data collection.

Pilot Study Procedure

The pilot study was conducted in a single hour-long session during the after school IT club session, where students would typically have used Scratch. The pilot procedure included the following steps:

1. Pre-study questionnaire (5 minutes)
2. Participants draw characters on paper (10 minutes)

3. Participants open DrawBridge, complete the pre-test (10 minutes)
4. Participants use DrawBridge and follow workbook (30 minutes)
5. Participants complete post-test (5 minutes)

Pilot Study Results

Two participants did not correctly follow drawing instructions given to them at the beginning of the session, resulting in incorrect segmentation of images and therefore missing or partially recognised characters in DrawBridge. The study overran by approximately 10 minutes due to students requiring longer than expected to complete questionnaire, assessment and drawing tasks.

A number of technical issues were identified during the pilot study:

- Students were not able to test animations in their web browser due to JavaScript execution restrictions (although still testable in DrawBridge).
- Students did not have write-access to the default DrawBridge save directory, which forced students to enter a new directory name.
- DrawBridge recorded separate student names for assessment and DrawBridge use, resulting in potential name ambiguity during analysis.
- Some scroll bars were not fully visible, making it difficult for students to scroll through visual blocks or text code.
- Some fonts were rendered incorrectly, resulting in some text that was difficult to read.

One participant had difficulty reading English after having just moved to the UK. The study required participants to read the provided worksheets. To address this issue, two live demos of DrawBridge were carried out to support the content written in the worksheets.

Changes to Procedure

The total time required for the pilot study indicated that the hour of allocated time was too short, and that sessions of 80 minutes or longer would be needed to allow enough time for participants to complete the required tasks. The procedure was adapted to reflect this, and modified to include an introductory discussion about programming, to improve participants' understanding of whether they had programmed before, and two demos, to help those students who could not follow the provided worksheets.

To avoid the incorrectly segmented drawings encountered by some participants, a list of guidelines was created to ensure characters would be recognised successfully (see Appendix D). Time was allocated at the start of the drawing session to explain the guidelines using an example drawing. Backup drawings were also supplied in case the student did not follow the guidelines.

8.5 Study 1: Quasi-experiment Design

A quasi-experiment was designed to further explore the research questions identified at the beginning of this chapter. The goal of this between-subjects study was to address the following questions: (RQ5) To what extent can motivation intervention features improve students' acquisition of Notational Expertise? (RQ6) To what extent do students from higher year groups improve more quickly than students from lower year groups? (RQ7) To what extent do students in higher year groups receive the same benefit from motivation intervention features as those in lower year groups? To address RQ5, the following sub-questions were explored (1) To what extent do motivation intervention features encourage students to move to abstract representations? (2) To what extent does an increase in students' use of abstract representations improve their acquisition of Notational Expertise?

DrawBridge was designed to be usable within a school environment. As with previous studies, this study was carried out during regular ICT/IT lessons in a conventional school classroom. During the study, integrated Adapted Parsons Problems were used as pre and post-tests to measure improvement in Notational Expertise. Measures of participant confidence were recorded via ratings given for each question answered in the assessments. DrawBridge instrumentation and questionnaire feedback was collected, as in previous studies.

To investigate the questions described above, a between-participants 2 x 2 Latin square design was used with two factors: (1) the inclusion of motivation intervention features, and (2) year group. Factor (1) consisted of a control: no motivation intervention features, and treatment: motivation features. Factor (2) consisted of Year 8 and Year 9.

8.5.1 Participants

Teachers interested in participating in the study were recruited during a Computing at School (CAS) workshop in Cambridge. The study took place at an independent school in Peterborough with mixed gender classes of Year 8 and Year 9 students. At the time of study, the school had recently achieved 83% grades A* - C in English and Maths GCSE, and was ranked second in the Peterborough area. On average, classes had 10 students; there were 25 female and 17 male participants in total.

Table 8.3: Latin Square Design

	Year 8	**Year 9**
Motivation Intervention	8MI (11/11)	9MI (8/11)
Standard	8S (8/9)	9S (9/10)

(Students attending for both lessons / Total number of students)

8.5.2 Procedure

The procedure was carried out with four classes taking two double lessons (eight double lessons in total), each lasting 1 hour 20 minutes. Each double lesson took place in the school's computing suite using high-specification 64-bit Windows 7 Intel Core i7 machines with 4GB RAM.

In (double) Lesson 1, participants were given a short introduction to the study and a pre-study questionnaire that collected basic information, such as age and name. The questionnaire also asked questions relating to previous use of computers, animation and programming (see Appendix D). After completing the questionnaire, participants were given 5 minutes to draw Halloween-themed characters on paper. The paper was collected and scanned while participants opened DrawBridge and started the Before-Lesson-1 assessment. On completion of the assessment, participants were given a short demo of Drawbridge and asked to work through the first DrawBridge worksheet. After 20 minutes, students were given a second demo that focused on animating characters. After 40 minutes in total using DrawBridge, worksheets were collected and participants were instructed to close the system to complete the After-Lesson-1 assessment. At the end of the lesson, local log data was collected from each computer.

In (double) lesson 2, participants spent 5 minutes drawing Christmas themed characters, which were collected and scanned, as in Lesson 1. Participants were then asked to complete the Before-lesson-2 assessment. After 10 minutes, participants were asked to read the first half of the Lesson 2 workbook, which explained variables and functions, followed by a short question-answer session about what students had read. Participants were then asked to read the second half of the worksheets, which described random numbers and how to generate them in DrawBridge, followed by a second question-answer session and a live demo showing how to add random numbers to the programming code used to run animations. After 40 minutes in total using DrawBridge, participants were asked to complete the After-Lesson-2 assessment. Finally, students filled out a post-study questionnaire that asked questions about what they liked and disliked about DrawBridge, and how it affected their interest for programming.

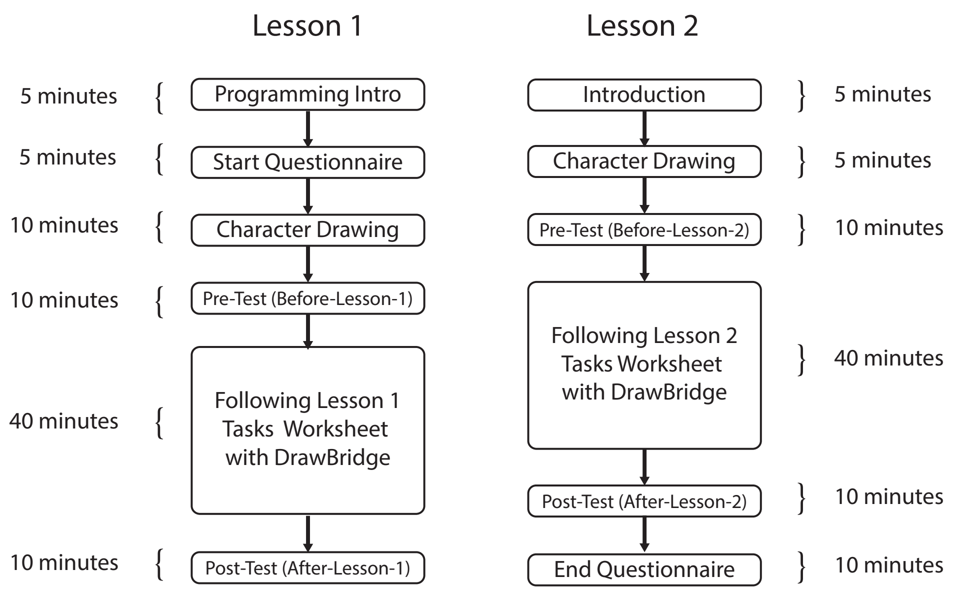


Figure 8.7: Procedure for Lesson 1 and Lesson 2

8.5.3 Data Collection

The pre-study questionnaire was adapted and improved from the questionnaire used in previous DrawBridge studies (see Chapter 6, and Appendix D). Compound questions, such as "Have you made animations before? If so, how?" were separated into explicit questions in order to reduce levels of non-answering. Participants were given the improved questionnaire at the start of the Lesson 1 (see Appendix D).

The post-study questionnaire distributed at the end of Lesson 2 was developed as an extension of the post-study questionnaire used in a previous study (see Chapter 6). The questionnaire asked participants to rate their enjoyment of DrawBridge, whether they would use it again, and the three best and three worst things about DrawBridge. In the improved questionnaire, participants were asked whether they thought DrawBridge helped them to learn about programming, how much they enjoyed using integrated assessments, how much they enjoyed drawing their own characters, and whether DrawBridge changed their interest in programming.

Adapted Parson Problem Assessments

DrawBridge was modified to include an integrated assessment tool that could be used to track participant progress at the start and end of each lesson during the study. The tool was created to use Adapted Parsons Problems, which were presented in Chapter 7 as an extension of Parsons Problems (Parsons & Haden, 2006), which have been successfully used to measure programming understanding at CS1 level (Denny et al., 2008). Rather than reordering full lines of programming code, as in traditional Parsons Problems, Adapted Parsons Problems measure Notational Expertise by allowing participants to reorder individual syntax elements from a given collection of elements. Evaluation and comparison of Adapted Parsons Problems, presented in Chapter 7, found that they provided a good proxy for code writing, and had a significantly higher response rate than code writing assessments. Assessment questions were chosen to cover a broad range of syntax and were based on scaffolding code used within DrawBridge (see Table 8.4).

Table 8.4: Before-Lesson-1 Assessment Answers

Assessment Answers	Syntax Elements							
	Brackets	Parameters	Semi-Colon	Assignment	Operators	Whitespace	Commas	Dot Syntax
`var myVariable;`			☑			☑		
`myVariable = 2;`			☑	☑		☑		
`myVariable = anotherVariable.name;`			☑	☑		☑		☑
`beginAnimation();`	☑	☑	☑					
`image1.setPosition(100, 200);`	☑	☑	☑				☑	
`myImage.setSize(340, 10);`	☑	☑	☑				☑	
`movingObject.stop();`	☑	☑	☑					☑
`myVariable = 2 + 2;`			☑	☑	☑	☑		
`myVariable = 2 * 2;`			☑	☑	☑	☑		

Parsons Marking Scheme

The marking scheme consists of two marking criteria: inclusion and distance. The inclusion mark is given for picking the correct syntax element required for the statement. The distance mark is given for arranging the elements in an order that minimises the distance to the correct answer. Both criteria are given equal weighting and are combined to give a total percentage mark for each question. The expected value differs for each question (depending on the number of required elements and distractors).

The order mark is calculated using the Levenshtein distance algorithm (Wagner & Fischer, 1974), which is a metric for calculating the distance between the candidate answer, and correct answer. More specifically, the number of insertion, deletions, or substitutions required. Table 8.5 shows the expected average mark for each question, calculated using a Monte Carlo simulation.

Table 8.5: Monte Carlo Simulation Expected Distance Values

Question	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
# Answers	3	4	6	4	9	9	6	6	6
# Distractors	3	4	2	3	2	4	2	3	3
Include (%)	50	50	75	57	81	69	75	67	67
Distance (%)	3	10.5	14.67	6.75	12	8	14.67	10.13	10.13
Exp Average	26.5	30.25	44.84	31.88	46.5	38.5	44.84	38.56	38.56

During the study, DrawBridge recorded log data both locally and via web analytics. Both sets of data were collected to avoid any loss of data due to Internet connection issues that might cause the analytics to fail.

8.6 Study 1: Results

Five participants of forty-one in total did not complete all assessments due to illness and extra-curricular activities. The data collected for these students is not reported during assessment analysis, but is used to describe questionnaire responses, when not related to assessment.

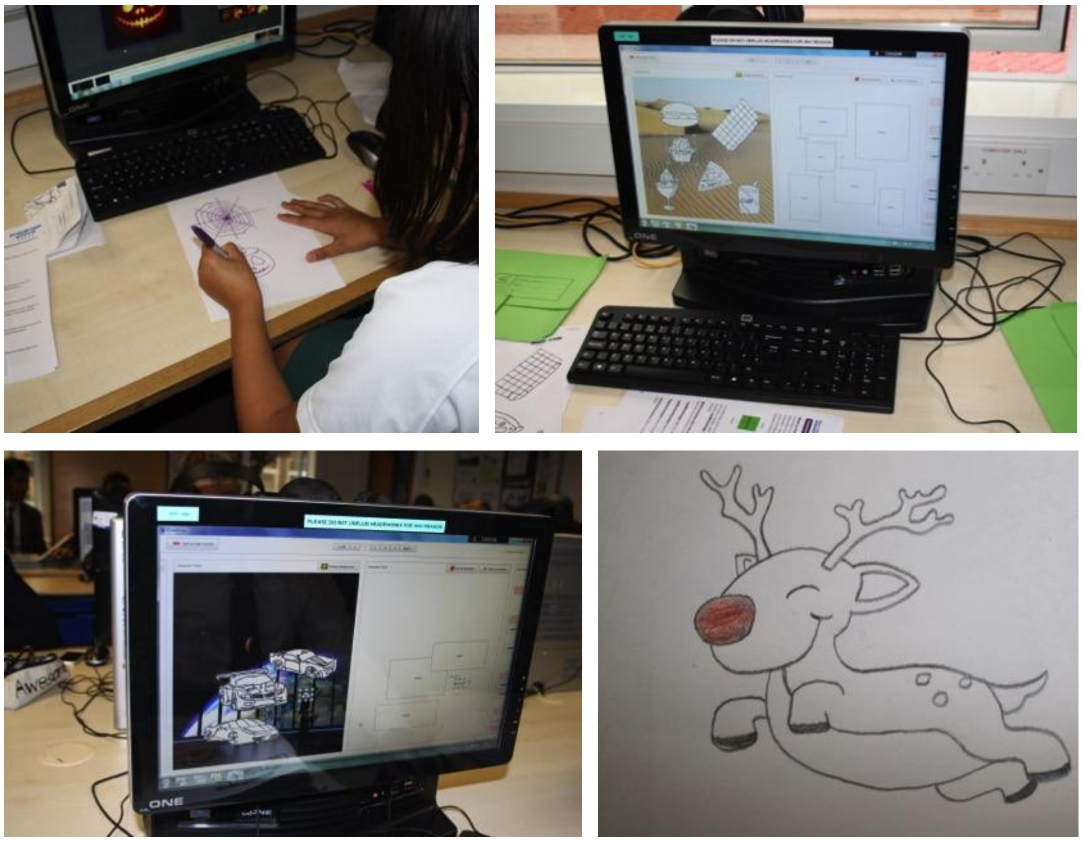


Figure 8.8: Examples of Animations Created in DrawBridge

(**Top Left/Bottom Right**: Student drawings of characters, **Top Right/Bottom Left**: Segmented characters with different backgrounds)

8.6.1 Assessment Performance

A Shapiro-Wilk test for normality was conducted to measure the distributions of results in each assessment. Of the four datasets collected: Before-Lesson-1, After-Lesson-1, Before-Lesson-2 and After-Lesson-2, all data were normally distributed except After-Lesson-1, $W = 0.8752, p < 0.0001$. Non-parametric tests for significance are used in analyses that include After-Lesson-1 data to ensure reliability of results.

Differences Before Using DrawBridge

To ensure fair comparisons, the Before-Lesson-1 performance of participants was compared by year group, and the inclusion of motivation intervention features. Year 8, achieved a mean of 55.42% (SD = 6.74), performing significantly better than Year 9, who had a mean of 51.11% (SD = 4.74), $t(31.03) = -2.26, p < 0.031$. This result supports anecdotal communication from the teacher, who suggested that in general Year 8 students were “geekier” than Year 9 and expected that they would improve at a quicker rate.

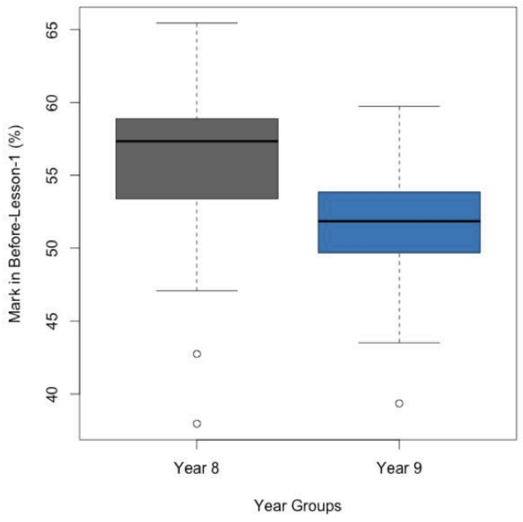


Figure 8.9: Differences between Year 8 and Year 9 before using DrawBridge

Absolute Score Comparison over Assessments

An ANOVA of student marks using three factors (year group, inclusion of motivation intervention features, and assessment) found a significant interaction, $F(3, 128) = 5.559, p = 0.0013$, suggesting that marks change significantly between lessons, year groups and with MI features.

A post-hoc Tukey test indicated that all groups except 8MI improved significantly between Before-Lesson-1 and After-Lesson-2:

- Year 8 Standard (8S): $p < 0.001$
- Year 8 Motivation Intervention (8MI):$p = 0.76$
- Year 9 Standard (9S): $p = 0.0004$
- Year 9 Motivation Intervention (9MI): $p < 0.001$

Figure 8.10 shows that in the After-Lesson-1 assessment, 8S scored highest (*Mean* = 76.54, *SD* = 14.14), 9MI scored the second highest (*Mean* = 61.10%, *SD* = 9.57), and 8MI (*Mean* = 52.44%, *SD* = 3.98) and 9S (*Mean* = 51.62, *SD* = 6.88) made minimal improvements.

Participants returned for Lesson 2 exactly a week after Lesson 1. Between both lessons, all groups except 8S improved slightly. The assessment scores after Lesson 2 show that 9MI scored a mean of 81.06% (*SD* = 9.56) and 8S scored a mean of 80.71% (*SD* = 12.47) performed similarly. 9S made large improvements scoring a mean of 69.65% (*SD* = 10.19) while 8MI made modest improvements scoring a mean of 62.16% (*SD* = 8.96).

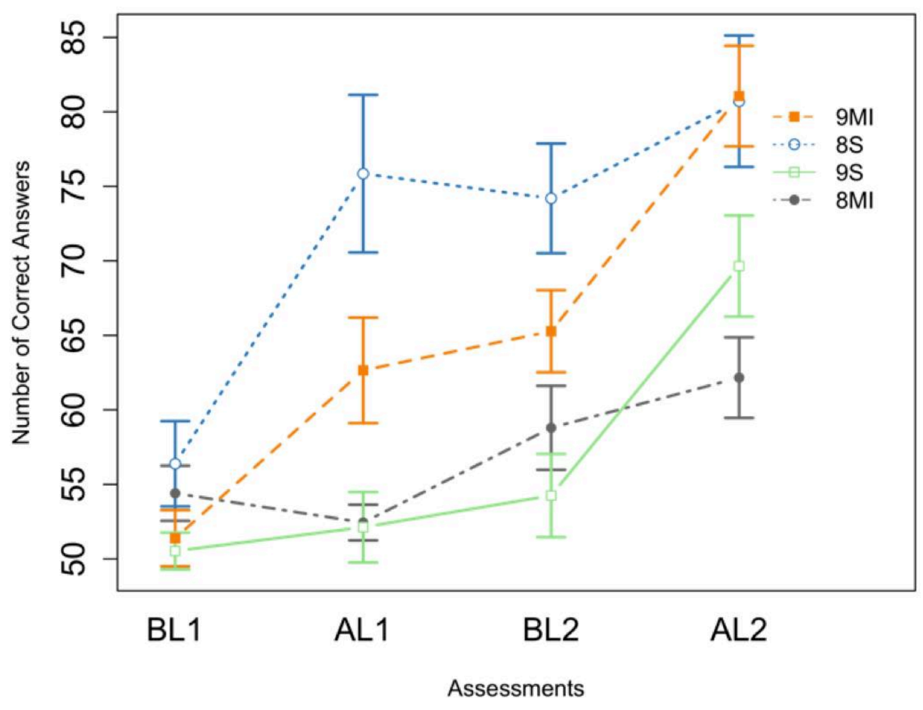
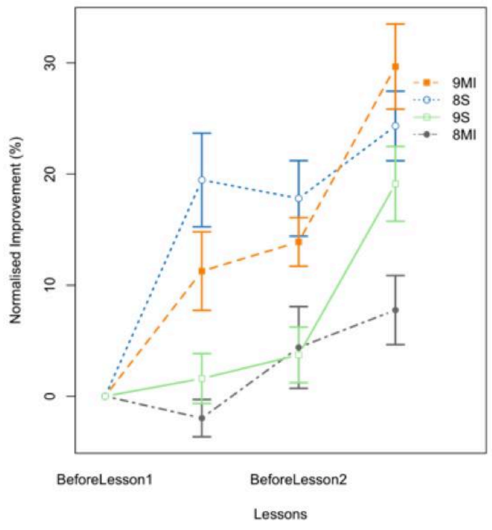


Figure 8.10: Student Marks Over Each Assessment

Gain Score Comparison over Assessments

As in previous studies, gain scores, calculated by subtracting the post-test and pre-test scores, are used to compare between groups over all assessments. The use of gain scores has been found to be a highly reliable measurement of change if variances are not equal (Dimitrov & Rumrill, 2003). Figure 8.11 emphasises gain scores by using scores normalised against Before-Lesson-1 scores.

Analysis of changes in Lesson 1 and Lesson 2 showed that 8S made the biggest improvement in Lesson 1, and was the only group to improve significantly ($p < 0.001$). In Lesson 2, 9S made a mean improvement of 15.40% (*SD* = 9.80), which was significant ($p = 0.021$). Group 9MI made a mean improvement of 15.78% (*SD* = 9.13), which was also significant ($p = 0.031$). Group 8S scored a mean improvement of 6.52% and 8MI scored a mean improvement of 3.37% (SD = 10.87).



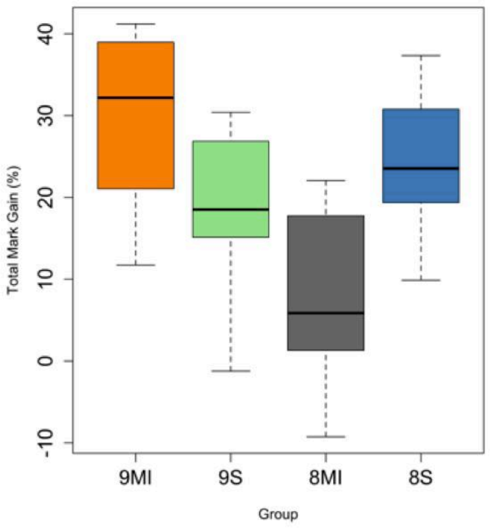


Figure 8.11: Normalised Assessment Results

Figure 8.12: Total Gain Over Both Lessons

The total gain scores, calculated from differences in After-Lesson-2 and Before-Lesson-1, found that 9MI gained the most marks with a mean score of 29.67% (*SD* = 10.84), 9S and 8S gaining less, and 8MI gaining the least with a mean of 7.76% (*SD* = 10.31).

An ANOVA found that there was a significant difference in gain scores between year groups over both lessons, with Year 9 improving by 24.08% on average, and Year 8 improving by 14.73%, $p = 0.027.$ There was no significant difference in total gain between groups with motivation intervention and those using the standard version of DrawBridge (Motivation Intervention *Mean* = 16.98 *SD* = 15.11, Standard *Mean* = 21.56, *SD* = 9.61). There was a significant interaction between Year Group and Motivation Intervention factors, $F(1, 32(= 16.02, p < 0.001.$

Table 8.6: Year 9 Mean Percentage Marks and Tukey Results

		9MI	**9S**	**Adj. P-Value**
Lesson 1	Before	**51.39** ± 5.34	50.53 ± 3.71	0.99
	After	**62.65** ± 10.01	52.12 ± 7.09	0.47
	Gain	**11.26** ± 9.17	1.59 ± 6.51	< 0.0001*
Lesson 2	Before	**65.28** ± 7.79	54.25 ± 8.38	0.39
	After	**81.06** ± 9.56	69.65 ± 10.19	0.34
	Gain	**15.78** ± 9.12	15.40 ± 9.80	0.43
Total	Gain	**29.67** ± 10.84	19.12 ± 10.1	< 0.0001*

(* significant at $p < 0.05$; ± standard deviation; highest in bold
total = After-Lesson-2 - Before-Lesson-1)

Table 8.7: Year 8 Mean Percentage Marks and Tukey Results

		8MI	**8S**	**Adj. P-Value**
Lesson 1	Before	54.41 ± 6.13	**56.39** ± 8.09	0.99
	After	52.44 ± 3.98	**75.85** ± 14.95	< 0.0001*
	Gain	-1.96 ± 5.57	**19.46** ± 11.92	< 0.0001*
Lesson 2	Before	58.80 ± 9.34	**74.19** ± 10.40	0.022*
	After	62.16 ± 8.96	**80.71** ± 12.47	0.0014*
	Gain	3.37 ± 10.87	**6.52** ± 6.43	0.99
Total	Gain	7.76 ± 10.31	**24.32** ± 8.84	< 0.0001*

(* significant at p < 0.05; ± standard deviation; highest in bold;

total = After-Lesson-2 - Before-Lesson-1)

Assessment Differences Over Year Group

A post-hoc Tukey test found significant differences between Year 9 group gains in Lesson 1: 9MI scored a mean gain of 11.26% (*SD* = 10.00) and 9S scored a mean gain of 1.59% (*SD* = 6.74), $(p < 0.0001($. There was no significant difference in gain scores between Year 9 groups in Lesson 2, with both improved by approximately 15%.

The test also identified significant differences between Year 8 group gains in Lesson 1, with 8S gaining a mean of 19.46 (SD = 11.92), and 8MI gaining a mean of -1.96 (SD = 5.57), $(\boldsymbol{p < 0.0001}($. The differences in gain scores for Year 8 groups in Lesson 2 were not significant.

Assessment Differences in Gender

A post-hoc Holm-adjusted pairwise t-test, conducted to investigate whether gender influenced assessment outcomes, found there was no significant difference in gain scores by gender, with males achieving a mean improvement of 22.83% (*SD* = 13.6) and females achieving a mean improvement of 15.69% (SD = 11.15), (p = 0.099). However, the gain scores for Males, with a mean improvement of 12.16% (*SD* = 12.55) in Lesson 1 were significantly higher than gain scores for Females, who had a mean improvement of 1.95% (*SD* = 8.76), $t(21.35(= 2.64, p = 0.015$. Figure 8.13 shows the mean gain scores over both lessons. Males outperform Females in all groups except 9S, which has the highest number of Females (8/10).

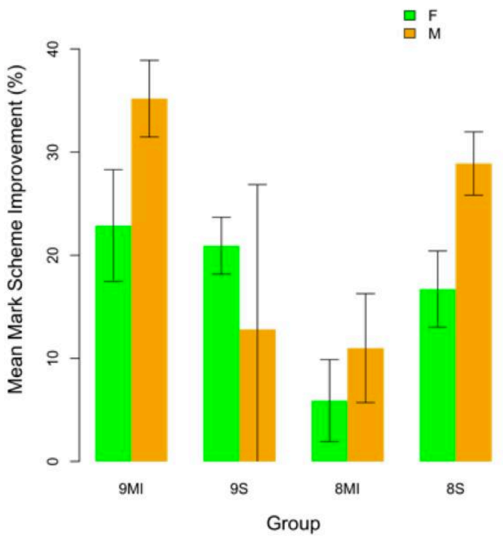


Figure 8.13: Marks for Females and Males in Each Group

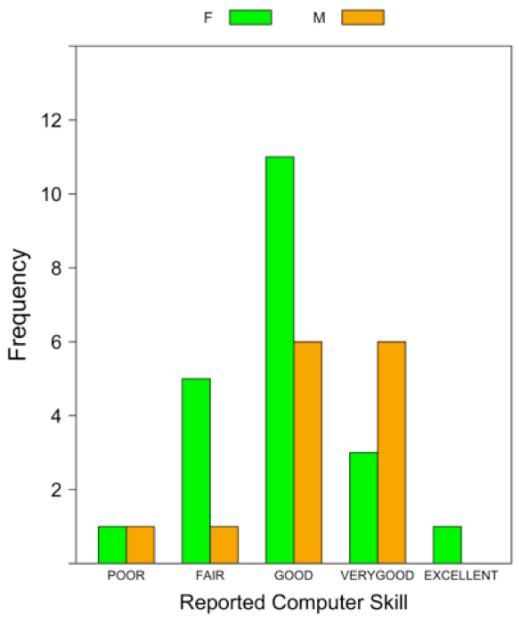


Figure 8.14: Distribution of Reported Computer Skill

Figure 8.14 shows that Females most commonly rated their computer skill as "Good" during the pre-study questionnaire. The distribution for Males appears to be shifted to the right, with equal numbers selecting "Good" and "Very Good" (see Figure 8.14).

8.6.2 Self-Efficacy of Participants

Self-efficacy, as described in Chapter 2, is defined as the justified belief about one's own capabilities to produce designated levels of performance (Bandura, 1997). The level of self-efficacy determines whether the participant might expend effort to complete the tasks, and for how long. It also determines whether the participant might use coping behaviour to get through the tasks (Bandura, 1977).

In this study, two metrics were recorded that can be used to measure changes in self-efficacy: self-reported computer skill, reported in the pre-study questionnaire; and confidence levels, reported for each question during assessments.

Self-reported Computer Skill

Participants were asked to select their level of computer skill in the pre-study questionnaire from a 5-point Likert scale, ranging from poor to excellent. A Kruskal-Wallis test was used to measure the interaction between reported skill and total gain results. Although the differences were not significant, $\chi^2(n = 36, 4(= 4.53, p = 0.34$, the median values generally correspond to reported skill (just one participant reported excellent computer skill).

- Excellent – 0.15 (n=1)
- Very Good – 18.67 (n=10)
- Good – 22.15 (n=20)
- Fair – 16.05 (n=8)
- Poor – 11.11 (n=2)

Assessment Confidence Levels

Every question answered in each assessment had an accompanying confidence rating that participants were asked to select. The confidence choices were "Right", "Seems Right Maybe", "Seems Wrong Maybe", and "Wrong".

A Kruskal-Wallis test of reported confidence over each assessment found that assessment mark differed significantly over confidence ratings in Before-Lesson-1, $\chi^2(n = 369, 3) = 9.18, p = 0.027$, After-Lesson-1, $\chi^2(n = 369, 3) = 14.95, p = 0.0019$, and Before-Lesson-2, $\chi^2(n = 325, 3) = 19.37, p = 0.00023$. However, there was no significant difference in scores relative to confidence levels in After-Lesson-2 $(P = 0.13($.

In the Before-Lesson-1 assessment, just a single participant used "Right" to rate the confidence in their answers. In the After-Lesson-1 assessment, 39 of 369 questions with a rating of "Right" scored a mean of 76.80% correct (*SD* = 24.05). In the Before-Lesson-2 assessment, 39 of 325 questions were answered with "Right" and scoring a mean of 81.66% (*SD* = 20.75). In the After-Lesson-2 assessment 60 of 325 questions were answered with "Right" and scored a mean of 80.42% (*SD* = 23.28). Those questions answered with a "Seems Right Maybe" rating scored a mean of 77.27% (*SD* = 19.65); those questions answered with a "Seems Wrong Maybe" rating scored a mean of 67.11% (*SD* = 21.02), and those questions answered with a "Wrong" rating scored a mean of 62.7% (*SD* = 20.18).

8.6.3 Time Use in DrawBridge

The Motivation Intervention version of DrawBridge was used by 8MI and 9MI (21 participants of 40 in total). In this study, four questions were investigated, including **one** concerning how well motivation intervention features encourage users to move to abstract representations.

The length of time spent viewing pairs of representations in DrawBridge was recorded during the study. Figure 8.15 shows the proportion of time spent viewing abstract panels (three pairs that include the visual block panel and code panel) and concrete (paper and Direct Manipulation) panels for each group during Lesson 1 and Lesson 2. Over both lessons, groups with motivation intervention features (8MI and 9MI) spent a larger proportion of time viewing abstract panels than their respective standard groups (8S and 9S).

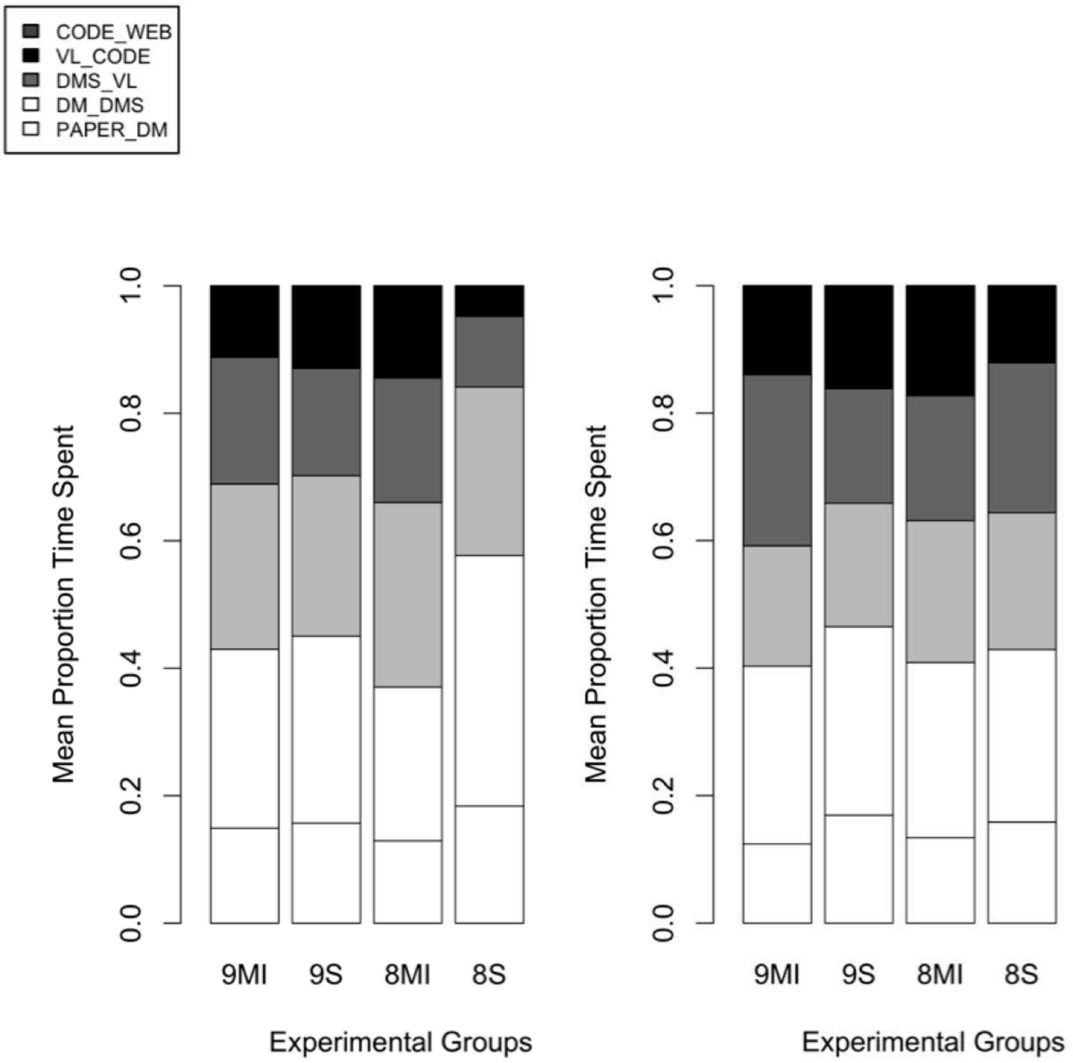


Figure 8.15: Time Spent viewing Code Representations
(**Left**: Lesson 1, **Right**: Lesson 2)

In Lesson 1, participants with motivation intervention features spent approximately 10 minutes more on average viewing abstract representations. A t-test found this to be highly significant (Motivation Intervention *Mean* = 22.34 mins, *SD* = 8.02, Standard *Mean* = 12.81 mins, SD = 5.72), $t(32.47(= -4.22, p < 0.001$. There was no significant difference between time spent viewing abstract representations in Lesson 2 ($p = 0.51$).

A measure of whether motivation intervention features improved participants' experience with DrawBridge is user feedback in the post-study questionnaire. 8 students of 21 mentioned coins as one of the three best things about DrawBridge. Just two participants mentioned it as one of the three worst things about DrawBridge. There was no significant difference between motivation intervention and standard group distributions for whether participants would use DrawBridge again. There was also no significant difference in how much both pairs of groups enjoyed using DrawBridge.

The first sub-question addressing RQ5 asks whether an increase in use of abstract representations would lead to improvements in Notational Expertise. A comparison between the proportion of time spent viewing abstract representations against gain scores in Lesson 1 and Lesson 2, (Figure 8.16), shows there is a noticeable separation between groups in Lesson 1, but an overlap in proportion of time spent on abstract representations in Lesson 2. A Pearson's correlation found no significant correlation between time spent on abstract representations and gain score in Lesson 1, $r(34(= 0.11, p = 0.53$ and Lesson 2, $r(33) = -0.10, p = 0.953$.

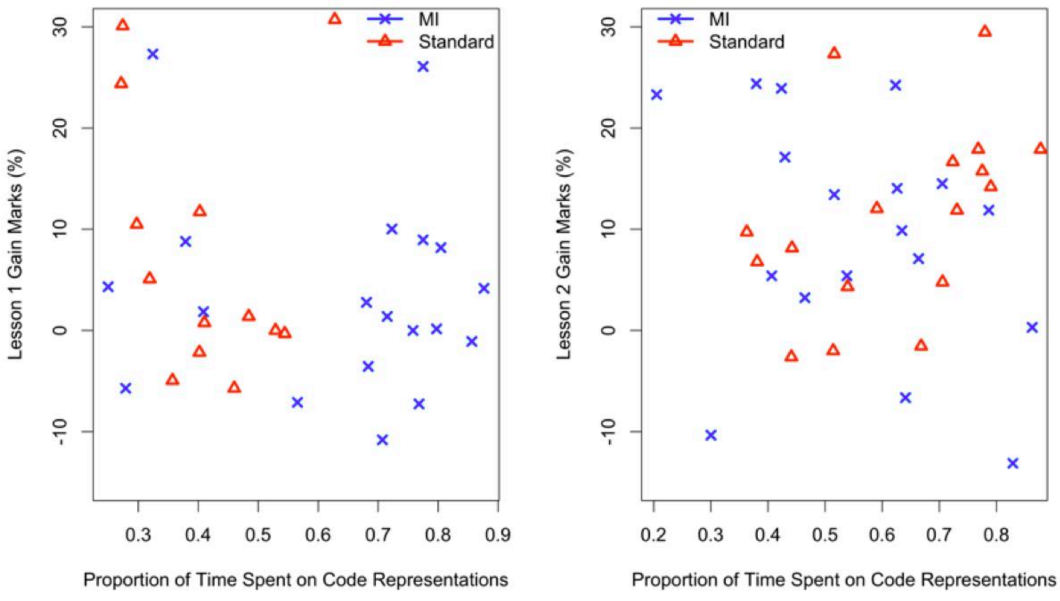


Figure 8.16: Proportion of Time Spent On Code Representations Relative to Lesson Results

Left: Lesson 1, **Right**: Lesson 2

Error results

Two kinds of errors were recorded for each participant via DrawBridge instrumentation: syntax errors, which relate to structural correctness of the program, and linter errors, which are based on static analysis of the program, and identify errors such as the use of uninitialized variables. On average, in Lesson 1, participants encountered more syntax errors (*Mean* = 21.16, *SD* = 35.58) than lint errors (*Mean* = 11.23 , *SD* = 21.37). In Lesson 2, however, there were more lint errors (*Mean* = 131.44, *SD* = 120.5) than syntax errors (*Mean* = 43.71, *SD* = 30.68). There was no significant difference between the errors encountered between groups in either lesson due to the high variance.

8.6.4 Questionnaire Responses

Pre-Study Questionnaire

Participants were presented with a questionnaire at the start of Lesson 1. When asked questions relating to their experience with computers, 35 of 41 students reported having created animations in the past, 24 reported having programmed before, and 9 reported having created a webpage before. When asked how they did these things, 34 students reported using Scratch for animations. Despite an explanation of programming before giving participants the questionnaire, just 21 of those 34 reported using scratch for programming. Of the 24 who had programmed, 22 said they had been for less than a year.

Participants were asked to report their level of skill using computers, three strengths using computers and three weaknesses using computers. The majority of students (38 of 41) reported having *fair*, *good* or *very good* skill; only 2 reported *poor* skill and 1 reported *excellent*. The most frequent strengths mentioned were Microsoft Word (7), the Internet (6), PowerPoint (6) and Scratch (4). The most frequent weaknesses were Microsoft Excel (8), programming (7) and databases (3). Overall, 24 students reported that they enjoyed using computers, while 11 were indifferent.

Post-Study Questionnaire

Students were given a post-study questionnaire (see Appendix D) at the end of Lesson 2 that contained six Likert scale questions and three free-response questions. 35 participants responded to the questionnaire. When asked whether they enjoyed using DrawBridge, 24 participants reported they “Liked it” or “Loved it” (see Figure 8.17-a). When asked whether they would like to use DrawBridge again, 26 participants said they would (see Figure 8.17-c). When asked whether it would be useful for the future, 26 said it would (see Figure 8.17-b). When asked if they liked using characters drawn on paper, 31 participants responded that they either “Liked it” or “Loved it”. More than half of participants (22) said that using DrawBridge had increased their interest in programming.

The post-study questionnaire also contained a question asking participants to rate how useful DrawBridge was for learning to program using a 5-point Likert scale ranging from “Really

Useless" to "Really Useful". Values of gain scores when related to the usefulness rating did generally correspond: those selecting "Useful" or "Really Useful" achieved a mean score of 21.57 ($SD = 13.09$). Those selecting "Useless" achieved a mean score of 14.87 ($SD = 10.79$). A Kruskal-Wallis test found that the difference was not significant ($p = 0.30($.

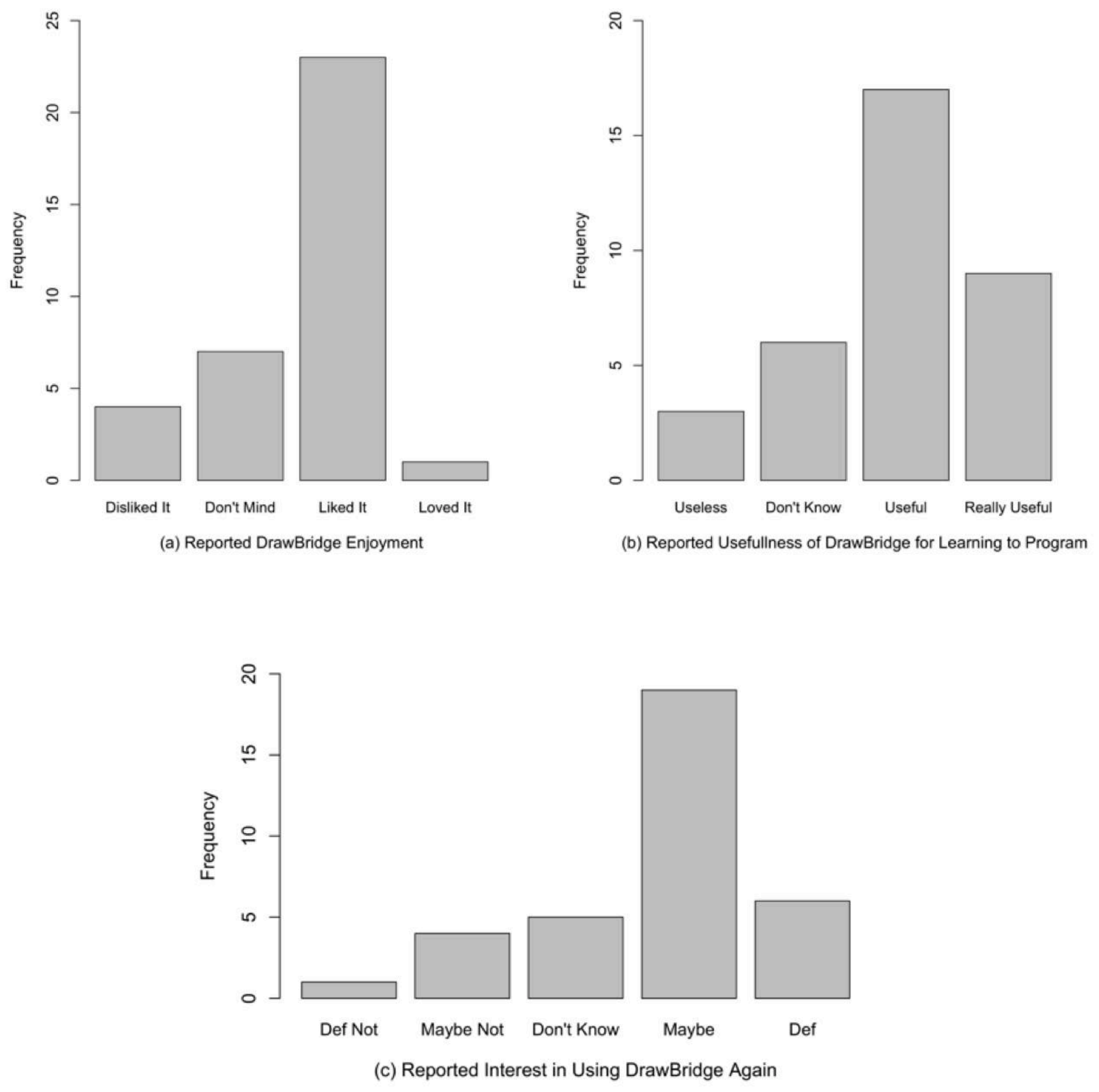


Figure 8.17: Post-Questionnaire Likert Ratings

(No students reported "Hated It" in (a), and no students reported "Really Useless" in (b))

8.7 Study 1: Discussion

The aim of this study was to address the following research questions: (RQ5) to what extent does the inclusion of motivation intervention functionality in DrawBridge increase the use of symbolic representations and therefore the acquisition of NE in novice programmers using DrawBridge; (RQ6) to identify the extent to which the year group of novice programmers using DrawBridge had any effect on acquisition of NE; and (RQ7) to investigate the extent to which the year group of novice programmers using DrawBridge affected the efficacy of motivation intervention features.

8.7.1 Effectiveness of Motivation Intervention Changes

Results showed that all groups except 8MI improved significantly between the first assessment and the assessment at the end of Lesson 2. The first question addressed in this chapter (RQ5) asked whether the addition of motivation intervention features would improve acquisition of NE. Results show that was no significant difference between groups with and without intervention features. However, 8S made a significant improvement in Lesson 1, and 9MI and 9S improved significantly in Lesson 2. The lack of a significant difference between groups with and without motivation intervention features suggests that motivation intervention features did not make a significant difference.

The first sub-research question of RQ5 asked whether motivation intervention features would encourage students to move to symbolic representations. Results from Lesson 1 of the study show that students who used the version of DrawBridge with motivation intervention features spent significantly more time using symbolic representations than students using the standard version. Although there is no significant difference in Lesson 2, the difference in Lesson 1 suggests motivation intervention features do help to encourage students to use symbolic representations.

The second sub-research question of RQ5 asked whether an increased use of symbolic representations would increase acquisition of NE. Analysis found no correlation between gain scores and time spent using symbolic representations, suggesting that increased use of symbolic representation does not increase acquisition of NE. This result is unexpected, but may due to lack of specificity in instrumentation data; time spent on pairs of representations is recorded, rather than a single representations.

The second question addressed in this chapter (RQ6) asked whether there would be a significant difference in acquisition of NE between year groups. In Lesson 1, there is no significant difference between year groups, while in Lesson 2, Year 9 improve significantly more than Year 8. The total gain scores show that over both lessons, Year 9 students performed significantly better than Year 8, suggesting that year group does make a difference, and that older year groups receive more benefit using DrawBridge. Before-Lesson-1 results show that there was a significant difference in performance between year groups, with Year 8 performing better than Year 9. The teacher also reported that, for some reason, the Year 8 group 8S was more interested in and adept at technical tasks than the others, which corresponds with results. Although attempts were made to gain access to previous student exam results to further investigate the differences in class ability, the school was not willing to provide this data.

The third question addressed in this chapter (RQ7) asked whether there would be a significant interaction between the two factors. Results show that there is a significant interaction between both factors in each lesson and overall, suggesting that the efficacy of motivation intervention features changes with year group. In the After-Lesson-2 assessment, 8MI achieved the lowest result, and 9MI achieved the highest. There are two conceivable explanations for this: (1) Motivation Intervention was beneficial to learning for Year 9 students, but detrimental to learning for Year 8 students, or (2) Group 8MI were lower ability or less engaged than the other groups.

8.7.2 Gender and Self-Efficacy in DrawBridge

Previous research suggests that males tinker more than females, particularly when tinkering has a low-cost (Beckwith et al., 2006). Males have also been found to display higher self-efficacy than females in complex computing tasks (Busch, 1995). DrawBridge allows users to begin using a "soft" style of working, similar to the Bricolage approach advocated by Papert and Turkle (Papert & Turkle, 1992), and then increase levels of "hard" thinking over time. In this study, males outperformed females on average, but there was no significant interaction between the total gain scores of either gender.

The measures of self-efficacy reported in the results include self-reported computer skill and assessment confidence levels. Self-reported computer skill shows that, students in general begin the process with a medium-high level of self-efficacy. Confidence ratings for each assessment question show that students' confidence more closely matches attainment over each assessment, suggesting that they are developing self-efficacy, or justified confidence.

8.7.3 General Experience with DrawBridge

Feedback from students in person for DrawBridge was very positive. The post-study questionnaire confirms this: most students found DrawBridge useful or really useful, reported that they liked or loved it, and would use it again. Students most commonly mentioned animations, using their images, and coins in feedback for the best things about DrawBridge.

8.8 Study 2: Quasi-experiment Design

To investigate questions that arose from Study 1, and to increase the external validity of results, a repeated study was carried out at a secondary state comprehensive school. The design of the study was identical to that of Study 1, but was carried out with 120 participants instead of 42, and took place in lessons that were 140 minutes instead of 120 minutes.

8.8.1 Method

The study took place in a mixed-gender, medium-sized state school in Cambridgeshire, UK. The study was largely consistent with Study 1, apart from three school-specific differences: (a) longer lesson times: 1hr 40m per double lesson compared with 1hr 20m for Study 1, (b) larger class sizes: up to 30 students, compared to an average of 10 in Study 1, and (c) lower specification computers: 32-bit Core 2 duo machines with 1GB RAM.

8.8.2 Changes to Procedure

The procedure for Lesson 1 was kept consistent with Study 1. The procedure for Lesson 2, however, was modified to remove the creation of new characters; instead participants were asked to reuse the characters created in Lesson 1. The change was a reaction to the increased time taken to scan all 30 participant images in Lesson 1, which took longer than expected (due to school infrastructure), reducing participants' time using DrawBridge.

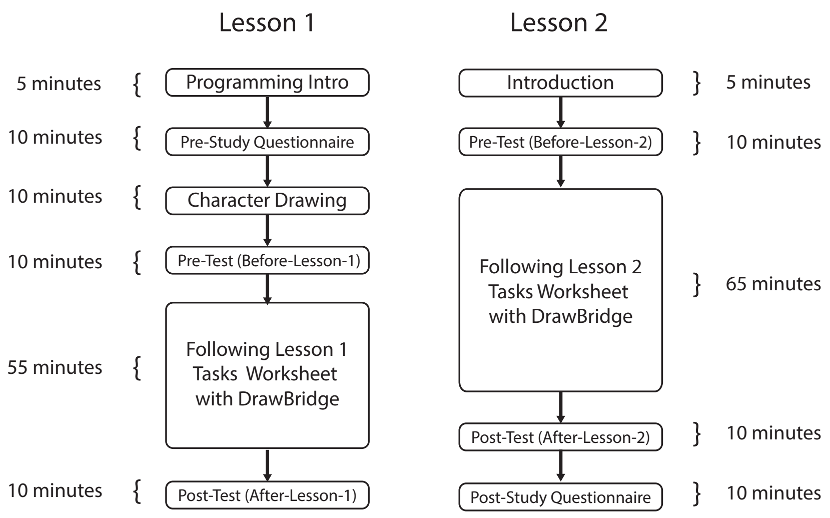


Figure 8.18: Study 2 Procedure

8.8.3 Participants

As in Study 1, two classes of students from Year 8 (aged 11-12), and two from Year 9 (aged 12-13) were recruited. In contrast to Study 1, however, the school separates its students into ability-dependent sets for Maths, and uses the same sets for Computing. All four classes were recruited from the medium-ability set in their year group, which were predicted to attain A* - B grades in their GCSEs.

8.8.4 Data Collection

The study used the same pre-study and post-study questionnaires, and the same assessment questions (see section 8.5).

8.9 Study 2: Results

Each class contained between 26 and 32 students. As in Study 1, each group had two double lessons, with (double) Lesson 2 occurring within a week of (double) Lesson 1. A proportion of participants' results were omitted from the assessment analysis, and sample size reduced to 45 participants, due to missing data as a result of technical issues with school machines, participant absence and incorrect naming (see Table 8.8). The criteria for assessment analysis were that assessment data was available for each of the four assessments. Data from participants who did not complete all assessments is used in analysis that does not relate to assessment (e.g. the pre-study questionnaire).

Table 8.8: Study 2 Group Latin Square Design

	Year 8	Year 9
Motivation Intervention	8MI2 (7/29)	9MI2 (10/26)
Standard	8S2 (15/32)	9S2 (13/29)

(Total participants with complete assessment data and total participant number shown in brackets)

8.9.1 Pre-Study Questionnaire

In total, 119 students responded to the pre-study questionnaire given at the start of Lesson 1. The majority of participants (65%) had created animations before but less than half (45%) of students reported they had programmed before, despite the teacher reporting that the school teaches SmallBASIC and had taught Scratch in past years. 37% students reported they had created animations with Scratch, but only 11% stated they had programmed using Scratch. The most frequently mentioned strengths were typing (11), using the Internet (9), and using Microsoft office packages (8). The most frequently mentioned weakness was programming (32). 61% of participants reported that their computer skill was good, very good or excellent and 27% reported having fair or poor skill. 61% participants reported having a good or high level of enjoyment, and 27% reported having an indifferent or negative enjoyment.

8.9.2 Assessment Results

A Shapiro-Wilk test found that, of all marks over both lessons, results collected in the After-Lesson-1 assessment were not normally distributed, $W = 0.94, p = 0.023$. The non-normal distribution in After-Lesson-1 appears to be due to Year 9 data (plotted as a histogram in

Figure 8.19), in which 3 participants achieved a score of 80% or more. Non-parametric tests are used to compare lesson data.

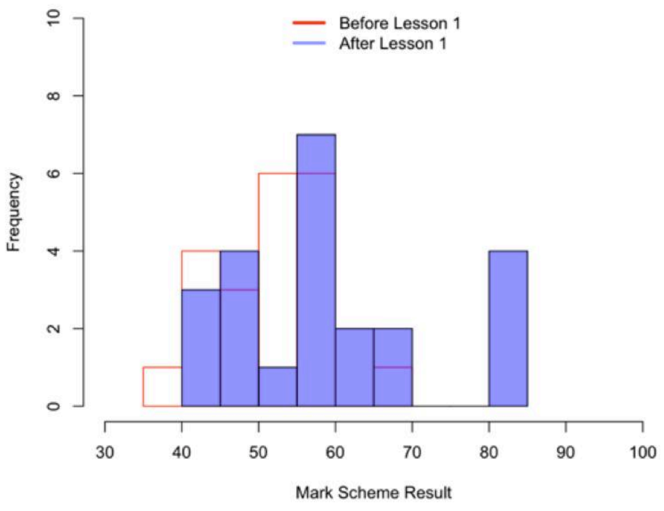


Figure 8.19: Comparison of Before and After Lesson 1 Distributions for Year 9

There was no significant difference between scores for each group at the start of Lesson 1; this was expected as no participants had any experience with JavaScript.

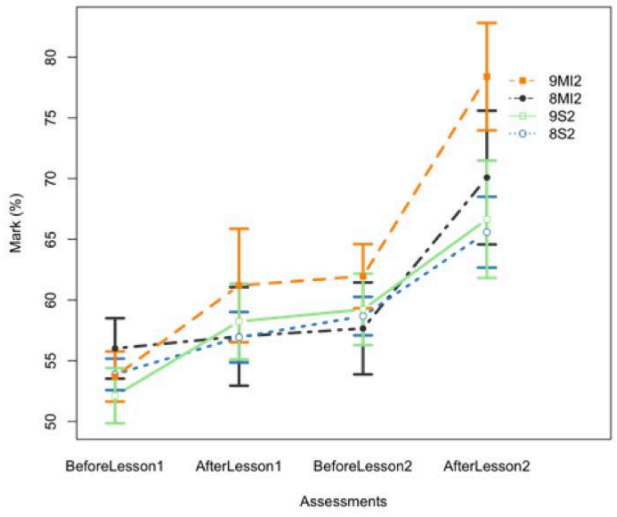


Figure 8.20: Mean Mark Scheme Results by Group

An ANOVA with three categorical factors (year group, use of motivation intervention features, and assessment) found no significant interaction between year, motivation intervention and assessment instance ($P = 0.87$). However, a post-hoc Tukey test found that groups in Year 9 improved significantly between Before-Lesson-1 and After-Lesson-2:

- 8S2: $t(14) = 3.72, p = 0.17$
- 8MI2:$t(6) = 3.07, p = 0.49$

- 9S2: $t(12) = 3.24, p = 0.048$
- 9MI2: $t(9) = 7.06, p < 0.001$

The assessment scores at the end of Lesson 2 show that both groups with motivation intervention features performed better than standard groups; 9MI2 performed the best (*Mean* = 78.3, *SD* = 14.0). 8MI2 performed second best, with a mean of 70.1% (*SD* = 14.6). Standard groups performed similarly, with Year 9 achieving a mean of 66.6% (*SD* = 17.42) performing marginally better than Year 8, who achieved a mean of 65.6% (*SD* = 11.29).

Gain Analysis

Figure 8.21 shows the gains over each assessment, normalised against the Before-Lesson-1 assessment. A post-hoc Tukey test found that no groups improved significantly in Lesson 1. In Lesson 2, 9MI2 improves near significantly ($p = 0.052$). The improvements in 9S2 and 8S2, although positive, are less than Motivation Intervention groups and not significant.

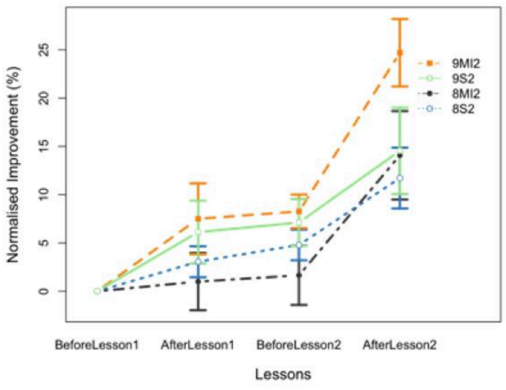


Figure 8.21: Marks Normalised against Before-Lesson-1 Scores

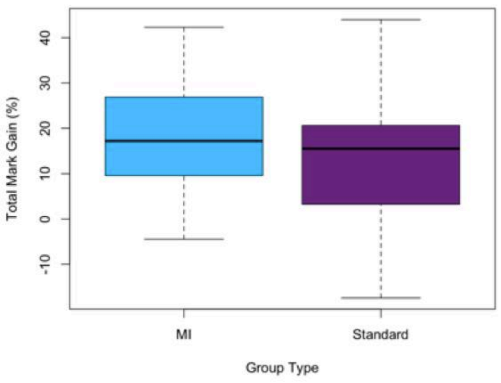


Figure 8.22: Overall Gains for Motivation Intervention and Standard Group Types

The largest total gain over both lessons was by 9MI2, who gained a mean of 24.70 (*SD* = 11.06). Groups 9S2 and 8MI2 improved by a similar amount, with a mean gain of 14.54, (*SD* = 16.16) and 14.07 (*SD* = 12.12) respectively. 8S2 made the least total improvement, with a mean of 11.71 (*SD* = 12.19). Figure 8.23 shows gains for each group over Lesson 1 and Lesson 2. An ANOVA found no significant difference between groups with intervention features (*p* = 0.13), no significant difference between year groups (p = 0.11) and no significant interaction between intervention features and year group (*p* = 0.35).

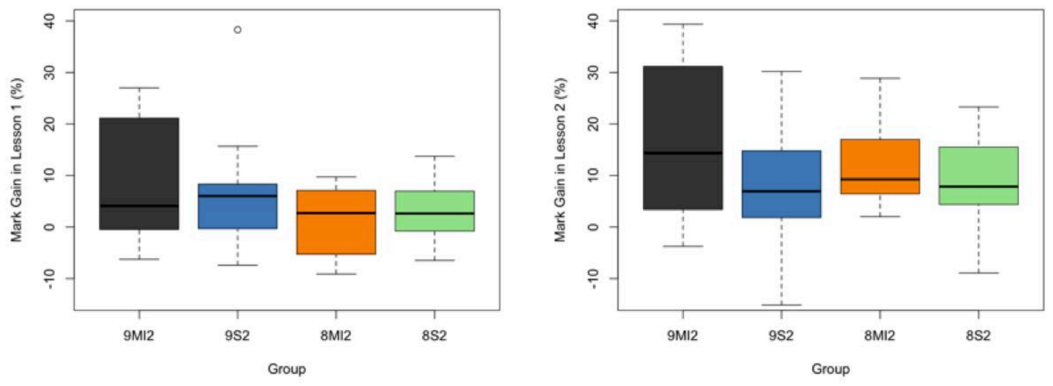


Figure 8.23: Assessment Gains

(Left: Lesson 1, Right: Lesson 2)

8.9.3 Gender

There was no significant difference in improvement between males and females in lesson 1. However, a holm-adjusted post-hoc t-test found that, in Lesson 2, females improved more than males, $(p = 0.0076($. The biggest improvement was found in group 9MI2, in which females improved significantly more than males, $(p = 0.020($. A t-test of the total gain scores over both lessons found that females improved significant more than males, $(p = 0.0087($. These differences are illustrated in Figure 8.24.

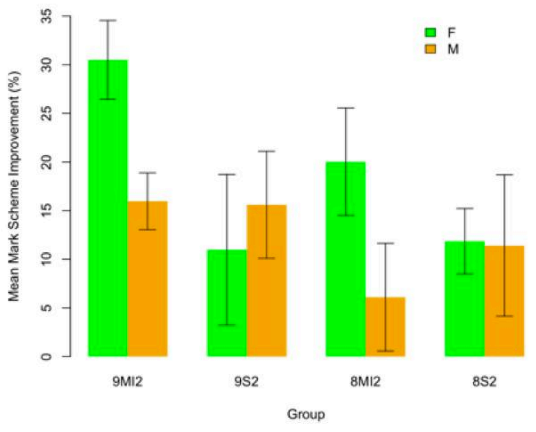


Figure 8.24: Gender Differences in Groups Over Both Lessons

(SEM Error Bars)

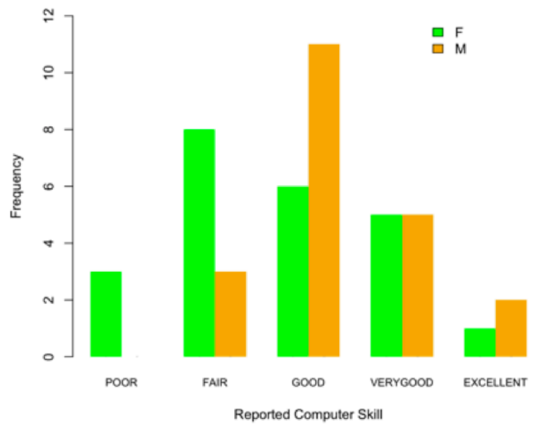


Figure 8.25: Gender Differences in Pre-Questionnaire Skill Reporting

Analysis of questionnaire results with gender found that there was no significant difference in reported computer skill between genders. Figure 8.25 shows that females may have had lower confidence than males, with 3 females reporting “Poor” skill compared to 0 males, and 5 males reported “Excellent” skill compared to 2 females.

8.9.4 Confidence Measures

As in the first study, this study contains two measures of participant confidence: self-reported computer skill and assessment confidence levels.

Self-Reported Computer Skill

A Kruskal-Wallis test of total gain scores over both lessons found no significant difference between gain scores relative to participants' self-reported computer skill ($p = 0.34($.

Assessment Confidence Levels

Table 8.9 shows the confidence distribution over each of the four assessments for all groups. Confidence increased during the sessions, and ultimately resulted in more than 65% of answers being classified as "Seems Right" or "Right". The high levels of confidence were not accurate, with just 18% of answers being fully correct at the end of Lesson 2. Confidence results are sensitive to repeated answering, where students select the same confidence level for each answer. For example, in Before-Lesson-2, 21 participants of 45 answered with the same confidence for every question. A chi-square test found that, over all lessons, there was a significant correlation between gender and confidence, $c^2(3, n = 1710) = 69.71, p = 4.94e^{-15}$.

Table 8.9: Participant Confidence Ratings

BL1 = Before-Lesson-1

	BL1 (%) n= 905	AL1 (%) n=806	BL2 (%) n=843	AL2 (%) n=751
Right	8.95	17.87	15.42	24.50
Seems Right	32.71	35.11	33.57	40.75
Seems Wrong	39.01	31.76	27.64	20.37
Wrong	19.34	15.26	23.37	14.38

8.9.5 Token/Coin Results

DrawBridge was instrumented to record the time spent viewing each pair of representation panels. The relative time spent viewing each pair of representations for each group in Lesson 1 and 2 can be seen in Figure 8.26. In both lessons, 9MI2 spent the most time viewing symbolic representations. There are only small differences between the time spent viewing symbolic representations for each group Lesson 1, compared to more pronounced differences in Lesson 2, where the groups with motivation intervention (9MI2 and 8MI2) spend more time with symbolic representations than groups using the standard version of DrawBridge. The difference was not significant (p = 0.28).

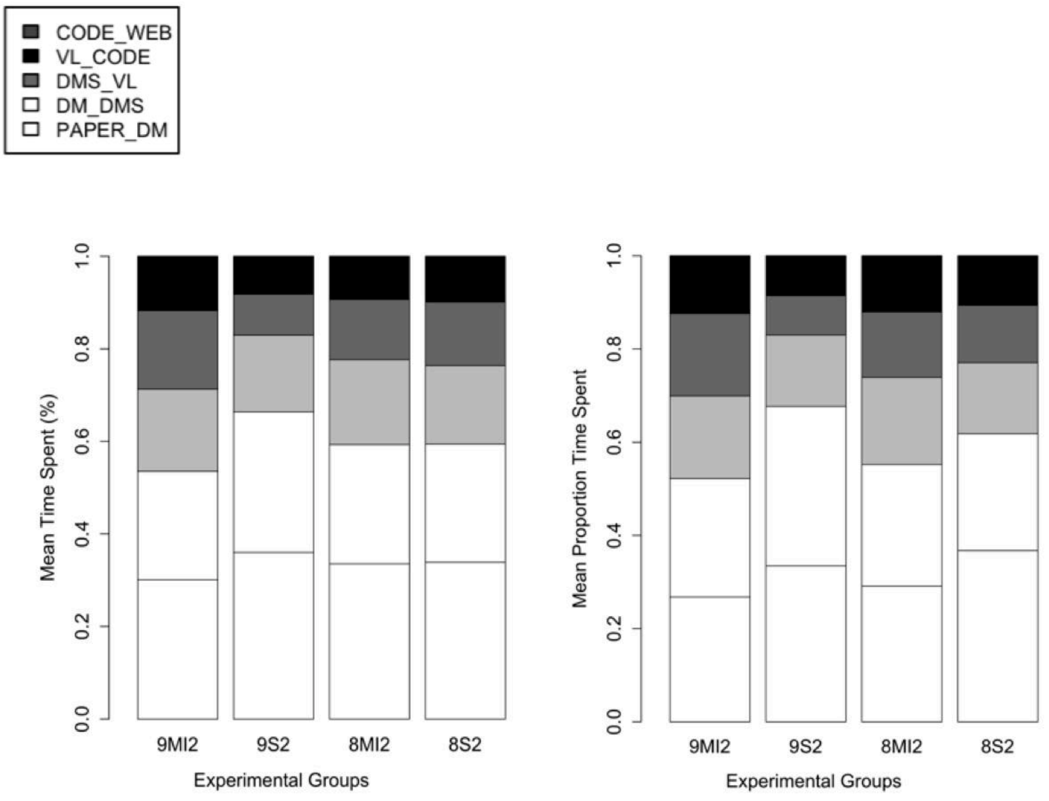


Figure 8.26: Mean Time Spent on Representation Pairs

(Left: Lesson 1, Right: Lesson 2)

In Lesson 1, participant learning gains appear to improve as the proportion of time spent on representations becomes equal (see Figure 8.27). A Pearson's correlation test found a significant positive correlation between proportion of time spent viewing symbolic representations and improvement in Lesson 1, $r(42(= 0.40, \mathrm{p} = 0.0069$, and a non significant correlation in Lesson 2, and $r(42(= 0.21, p = 0.17$ respectively.

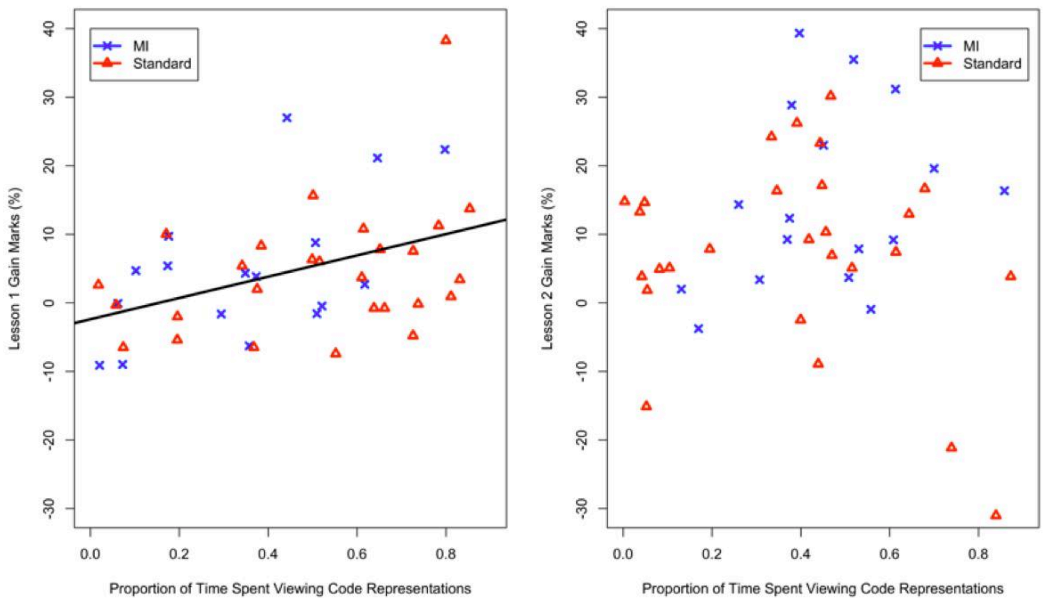


Figure 8.27: Proportion of Time Spent Viewing Code Representations Relative to Lesson Improvement

(**Left**: Lesson 1 with least squares regression line, **Right**: Lesson 2)

Of the 55 participants in groups with motivation intervention features, 8 reported that coins were the best things about DrawBridge, writing "coins, different panels helped a lot" and "being able to use your own drawing and the coins". 3 participants thought coins were the worst things about DrawBridge, writing "scanning of images makes using it very difficult, coins system can annoy", "coins, codes, limited amount of pictures", and "coins, slow, glitchy".

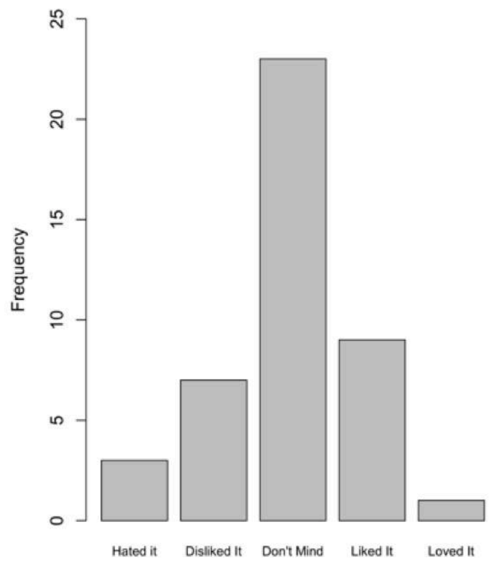
Error Results

DrawBridge instrumentation provides details of syntax errors, which highlight problems with structure, and linter errors, which report problems detected during static analysis (e.g. variable not declared). There was a non-significant correlation between the total number of linter errors and total improvement, $\mathrm{r}(39) = 0.18, p = 0.24$. On average, over both lessons, students made 90.0 syntax errors, and 66.0 lint errors. An ANOVA found there were no significant differences between the frequencies of syntax errors and linter errors between all groups ($p = 0.65$ and $p = 0.32$ respectively).

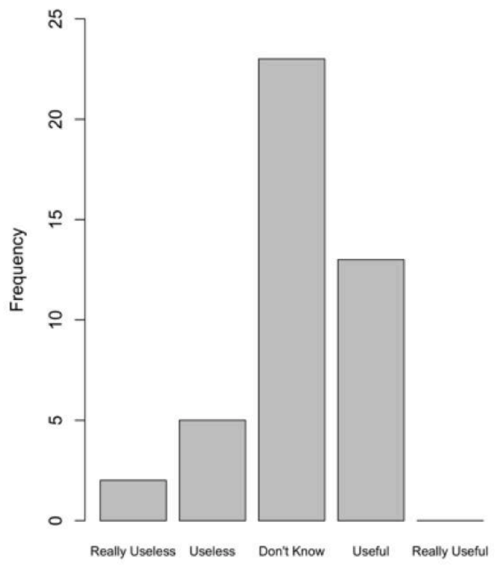
8.9.6 Post-Study Questionnaire

As in Study 1, the post-study questionnaire contained questions asking participants to rate how useful DrawBridge was for helping them to learn about programming (see Appendix D). A Kruskal-Wallis test found a non-significant difference between the total gain scores of each group selecting different answers, $\chi^2(n = 47, 4) = 8.37, p = 0.079$: Students selecting "Really Useless" or "Useless" scored a mean of 8.67% (*SD* = 16.97), improving less than those selecting “Useful”, who scored a mean of 24.96% (*SD* = 10.73). Of a total of 47 responses, 13 students reported that DrawBridge was useful for learning to program, 23 were unsure, 7 students reported it was not useful and 4 did not answer.

When asked how much they enjoyed using DrawBridge, 33 of 46 participants reported they didn’t mind, liked or loved DrawBridge (see Figure 8.28-a). When asked if they would use it again, 25 of 46 said they might (see Figure 8.28-c). When asked how much they enjoyed the integrated assessments, 21 of 46 participants were indifferent, while 13 disliked it. When asked how much they enjoyed using their own characters, 40 of 46 participants didn’t mind, liked or loved it. Finally, 13 of 46 participants said that DrawBridge had increased their interest in programming (see Figure 8.28-b).



(a) Reported DrawBridge Enjoyment



(b) Reported Usefullness of DrawBridge for Learning to Progran

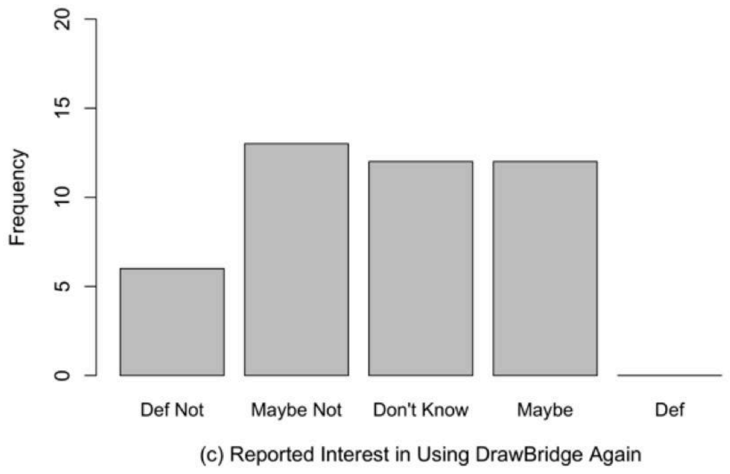


Figure 8.28: Post-Questionnaire Likert Question Responses

When asked to list the three best about DrawBridge, 50 participants referred to drawing their own characters, 41 referred to creating animations, 17 referred to enjoyment in learning to code, and 14 referred to the system's ease of use. When asked to list the three worst things, 31 participants referred to programming.

8.10 Study 2: Discussion

This section discusses the results of the second study to investigate the effectiveness of motivation intervention features for improving Notational Expertise. Participant self-efficacy, gender differences and error analysis are discussed, followed by the study limitations.

8.10.1 Effectiveness of Motivation Intervention Changes

The results of the study show that all groups improved significantly between the first assessment and final assessment at the end of Lesson 2. Results also show that groups using the version of DrawBridge with motivation intervention features improved more than those without overall. Group 9MI2 improved the most in Lesson 1, and both 9MI2 and 8MI2 improved significantly in Lesson 2. 9MI2 and 8MI2 also ended Lesson 2 achieving the highest marks on average (78% and 70% respectively).

The first question addressed in this chapter (RQ5) asked to what extent motivation intervention features would improve acquisition of NE. Although over both lessons, an ANOVA found that the difference between groups with intervention features and those without is not significant (p = 0.134), differences in means suggest that the features did improve NE acquisition. Technical issues such as problems with Internet connections, incorrect versions of Java, and broken peripherals resulted in a proportion of students being slowed down, distracted or having assessment results omitted from the study. These issues may have been responsible for the lack of any significant improvement due to motivation intervention features.

The first sub-research question of RQ5 asked the extent to which motivation intervention features would encourage students to use symbolic representations more than those without the features. Analysis of instrumentation data found that, although there was a visible difference in the mean time spent viewing symbolic representation in Lesson 2 (Figure 8.26), there was no significant difference in proportional use of symbolic representations, suggesting that MI features did not encourage students to use symbolic representations for a significantly longer time.

The second sub-research question of RQ5 asked to what extent an increase in the use of symbolic representations would increase student acquisition of NE. Analysis of results from Lesson 1 found a significant positive correlation between the time spent viewing symbolic representations, and improvement in NE. However, there was no correlation in the second Lesson, which suggests that other factors influenced students' acquisition of NE, suggesting that an increase in use of symbolic representations does not directly increase students' acquisition of NE. Intriguingly, in Lesson 2, it appears as though there is a weak trend towards *equal* use of symbolic and non-symbolic representations resulting in the biggest improvements. These results indicate two possibilities: the time viewing pairs of representations may not provide a good measure of how students learn using the tool, and the learning process may be too complex to model using any measure other than integrated assessment mechanisms.

The third question addressed in this chapter (RQ6) asked to what extent year group would affect acquisition of Notational Expertise. Year 9 groups improved more than Year 8 on average in both lessons. However, the difference was not significant, suggesting that DrawBridge works equally well for students in lower year groups. However, although the difference was not significant in Study 2 ($p = 0.11$), it was significant in Study 1 ($p = 0.027$), with mean total improvements in year groups being comparable in both studies, suggesting that there is a difference between acquisition of NE between year groups.

The fifth question addressed in this chapter (RQ7) asked whether there would be a significant interaction between use of motivation intervention features and year group. In contrast to Study 1, which found a significant interaction, Study 2 found a non-significant difference (p = 0.35), suggesting that the efficacy of motivation intervention features may be linked to year group.

8.10.2 Confidence, Self-efficacy & Gender

The majority of participants reported high levels of computer-related confidence in the pre-study questionnaire. However, analysis showed there was no relationship between reported computer skill and assessment results. Levels of confidence grew after each lesson from 41% of questions being rated "Right" or "Seems Right" in Before-Lesson-1, to 65% in After-Lesson-2.

The results regarding the effects of gender on the use of DrawBridge show that females in groups with motivation intervention features (9MI2 and 8MI2) improved significantly more than males despite a close to 50/50 gender balance in both groups. The difference could have been because females respond better to gamification features (Koivisto & Hamari, 2014), or may have been due to gender bias during the study (e.g. in instruction materials), bias in DrawBridge itself, or external factors such as classroom behaviour. Analysis of results in Study 1 found no significant difference between female and male performance, suggesting that external factors are more likely to be the cause. Anecdotal evidence from observations suggests that male students were more disruptive than female students during the study, which could explain the difference.

8.10.3 General Experience with DrawBridge

The pre-study questionnaire provided further evidence to suggest that students in both Year 8 and Year 9 were not aware they had been programming when using systems like Scratch, despite an introductory discussion on programming.

The post-study questionnaire results were mixed; most students were positive or indifferent about their experience with DrawBridge and 57% of students either liked or loved the use of their own characters during the process. During the pre-test questionnaire, several students wrote that one of the worst things to do with computers was to use them for homework/exams. It was therefore unsurprising that 28% of students stated they disliked the integrated assessments in the post-study questionnaire.

The school machines used in the study were low-specification, which resulted in DrawBridge slowing down under large amounts of animation. Some students referred to this in their post-study questionnaire feedback.

Despite the motivation intervention system working well for the majority, two students in Group 8MI2 attempted to subvert the system by closing and restarting DrawBridge to reset the number of coins available. Instrumentation shows they did this a total of 18 times. This data was omitted from the final results.

8.11 Design Implications for Drawbridge and Other MERs

As in Chapter 6, observations and the analysis of experimental results presented in this chapter have resulted in implications for the design of DrawBridge and other MER systems.

Gamification and MERs

The motivation intervention features described in this chapter were designed using principles of gamification and Design with Intent (DwI). Observations during each experiment confirmed that students quickly understood the coin mechanism, and the way in which coins were transferred between representations. However, the coins mechanism had the goal of encouraging students to interact with symbolic representations, and was therefore not closely bound to student understanding. Although results from Study 2 suggest that increased interaction with symbolic representations does improve student acquisition of RE, results in Study 1 do not, suggesting that gamification features more closely linked to acquisition of RE may provide students with more benefit. One potential design might be for students to answer questions such as those included in the DrawBridge assessment. Giving correct answers, writing a working program to complete a pre-specified task, and successfully debugging code may "unlock" new features.

Guiding Representation Transition

Existing MER systems, such as the Leogo and Visualise systems described in Section 2.4.3, allow users to have instant access to all available representations. The design of DrawBridge paid particular attention to reducing the number of representations available to users in an effort to reduce cognitive load, and provide representations appropriate to the students' programming experience. Observations and experimental results presented in this chapter, and Chapter 6, found that students skipped ahead to explore all representations before interacting with them sequentially. To avoid this kind of distraction, and to more closely guide students' progress, design improvements such as hiding later representations and "unlocking" them only when students have achieved a particular level of ability, could be implemented.

8.12 Conclusions

Two studies were carried out to investigate the use of motivation intervention features in DrawBridge through the use of novel integration assessment features. The studies took place in two very different school environments: an independent school with small class sizes, and a state school with large class sizes. Results from both studies showed that all groups of students improved between the assessment at the beginning of Lesson 1 and the assessment at the end of Lesson 2. Results from the first study showed that overall, Year 9 students improved significantly more than Year 8, and that there was a significant interaction between the factors showing that acquisition of NE increases with motivation intervention features and an increase in year group. The second study also showed that Year 9 improved more than Year 8, though the differences were not significant. In both studies, the Year 9 group with motivation intervention features improved most in each lesson and overall. The Year 8 group with intervention features improved least during Lesson 1 in both studies, suggesting that Year 8 students may have required longer to master motivation intervention features in DrawBridge, possibly because of the added complexity. However, students in Group 8MI2 (Study 2) made a significant improvement in Lesson 2, suggesting that longer lesson times (20 minutes extra), may give students the time needed to become familiar with complex features.

The design and evaluation of motivation intervention features, and investigation of questions during this chapter contributed towards three of the research questions addressed in this thesis. The first research question (RQ5) relates to how well the addition of motivation intervention features encourages users to move to abstract representations. Although the results do not give a definitive answer to this question, they do strongly suggest that intervention features improve acquisition of NE for older students. In particular, results in Study 2 found a positive correlation between the time spent viewing symbolic representations and acquisition of NE. In addition, observations and questionnaire feedback suggest that MI features may have provided some "Useful Awkwardness" (Blackwell et al., 2001) for students, which increased the amount of reflection, and thus allowed them to spend their coins more wisely.

Two sub-research questions that addressed RQ5 were concerned with the extent to which motivation intervention features increase the use of abstract representations, and the relationship between the time spent using abstract representations and acquisition of NE. In contrast to expectations, results from Study 1 found no significant correlation between the proportion of time spent viewing abstract representations and total gain score. However, results from Study 2 found a significant positive correlation between the proportion of time and total gains score, indicating that use of symbolic representations and acquisition of NE are linked. For the first study, it is possible that the time data used for this analysis was not fine-grained enough to be useful. It is also likely that this measure may have been too simplistic to model the complex learning process students go through when using DrawBridge.

The second research question (RQ6) was concerned with whether the acquisition of Notational Expertise was dependent on student year group. Study 1 shows there was a significant difference in improvement of Notational Expertise between Year 9 and Year 8, and that this was likely due to an interaction between the two factors. Results suggest that older students are able to use DrawBridge successfully with motivation intervention features, but younger students may benefit from using the standard system for a longer period before moving to abstract representations.

In addition to the three main research questions, this chapter investigated gender differences in the use of DrawBridge, confidence and resulting self-efficacy in each study, and the effect of errors on Notational Expertise. Results from both studies show there was no significant difference in total gain scores between genders, indicating that DrawBridge is useful for both males and females. Confidence scores in each assessment question show that confidence levels improve with increased use of DrawBridge. Error analysis found no correlation between frequency of errors and improvement, suggesting that participants' reaction to the errors are likely to be more important than the errors themselves. Integrated assessment and analytic results found that, although student feedback on the assessments in both studies was indifferent on the whole, all students (who did not experience technical issues) completed the assessments, allowing direct comparison of progress over each lesson between groups.

Finally, two design implications related to guiding students' transition between representations, and improving motivation intervention features were presented, with recommendations for DrawBridge and other MERs based on observations and experimental analysis.

Chapter 9 Conclusion

Overall Research Goal:

To what extent can Multiple External Representation systems (MERs) be used to improve student acquisition of Notational Expertise (NE)?

This thesis explored how Multiple External Representations (MER) can be used to improve acquisition of Notational Expertise (NE) in educational programming environments used by students in Key Stage 3 (aged 11-14). The thesis began by presenting a review of the theories of cognitive development, knowledge transfer, fragile knowledge and related curriculum issues associated with the development of educational programming environments. A review of existing educational programming tools, paying particular attention to usability was also presented, categorised by the type of representation used in the tool and their usability trade-offs, followed by a review of MERs and their use in several domains. Chapter 3 presented Modes of Representational Abstraction (MoRA), an analogical framework that can be used to classify representation use in both mathematics and programming. It also presented a definition of Notational Expertise (NE), a collection of mental competencies that allow students to identify and translate concepts between representations. Chapter 4 described teaching interviews that identified existing teaching strategies that corresponded with representation transition strategies generated from the MoRA framework. Chapter 5 used the strategies identified using the MoRA framework to describe the design and implementation of DrawBridge, a prototype educational programming tool using MERs. Chapter 6 described a study that investigated the practical success of the strategies identified in previous chapters in order to guide future design decisions. Chapter 7 presented a study to identify and compare assessments measuring NE. Chapter 8 presented a two-part study that investigated the efficacy of motivation intervention features added to DrawBridge, and compared use of DrawBridge between year groups.

This chapter will present the findings made throughout the thesis to address the research questions described in Chapter 1, which will be used collectively to address the research goal. A discussion of the contributions of this thesis is then presented, followed by the limitations of this work and future work.

9.1 Summary of Findings

To what extent can Multiple External Representation systems (MERs) be used to improve student acquisition of Notational Expertise (NE)?

Findings from quasi-experiments presented in Chapter 6 and Chapter 8 have suggested that the use of MERs in educational programming languages can be used to significantly improve acquisition of NE in students in Key Stage 3 (age 11 – 14) when appropriate representation transition strategies are used. Results measured using an assessment type designed specifically to measure Notational Expertise, Adapted Parsons Problems (APPs), found that all groups improved between the assessment at the beginning of their students' first lesson using DrawBridge, and the assessment at the end of the final lesson. Measures of confidence throughout each assessment found that students' confidence increased over time. Additionally, students responded very positively to motivation features for DrawBridge, and reported an increased interest in programming in the future.

The Order of Symbolic Representations:
Can the order of symbolic representations (Visual-First and Text-First) affect students' ability to acquire Notational Expertise in MER systems?

The MoRA framework defined in Chapter 3 helped to identify representation transition strategies that could be used to develop NE and allow students to move between low-abstraction tangible representations to unlimited-abstraction text representations. DrawBridge, a prototype educational programming environment, was created to investigate the effectiveness of MERs transition strategies. Chapter 6 presented several quasi-experimental studies that investigated the representation transition strategies identified in the MoRA framework. A comparison between groups using different orders of symbolic representations in DrawBridge found that there was a non-significant difference in improvement between groups using a visual-first order of representations and groups using a text-first order. However, differences in mean improvement suggested that visual-first groups improved more than text-first groups, leading to use of the visual-first strategy in DrawBridge for future studies.

These findings provide validation for the use of the main strategy identified in the MoRA framework, and supports Stenning and Oberlander's assertion that graphical representations limit abstraction, making them easier to compute than unlimited abstract representational systems, such as text. It also validates teachers' existing strategies discussed in Chapter 4, and curriculum changes discussed in Chapter 2, which suggest that it is preferable to use visual languages before text languages.

The Need for Low-Abstraction Representations:

To what extent do low-abstraction representations improve acquisition of NE in MER systems?

Findings from studies presented in Chapter 6 did not identify a significant difference in improvement between groups with and without the use of low-abstraction representations. However, deficiencies in the pre/post-test assessment resulted in significantly different pre-test results for some groups and low response rates for a section of each assessment. Follow-up questionnaires found that low-abstraction representations – in this case, physical paper where students could draw and edit their own characters, and Direct Manipulation representations, which allowed students to create animations using Programming by Example (PbE), were frequently mentioned as the best things about DrawBridge, and provided major motivation for students. Finally, DrawBridge instrumentation data showed that students using versions of DrawBridge without low-abstraction representations spent ten times as long viewing symbolic representations compared with those using versions of DrawBridge with low-abstraction representations.

Although it was not possible to directly compare the motivations of students with and without low-abstraction representations, observations suggested that students with low-abstraction representations were more motivated to use the tool, and more likely to want to use it in future. The use of Computer Vision techniques to automatically segment hand drawn characters to provide a basis for programming offered a unique, "hand crafted" motivation for programming, which could not be easily replicated elsewhere. This kind of motivation takes an alternative approach to that of popular educational programming systems, which have been criticized for several reasons (e.g. for providing a user interface that enables the creation of games that are not of the standard that students expect in the 21st century (MacLaurin, 2011)). It also further supports the use of tangible representations, such as robots and textiles, as an initial representation that can provide motivation for educational programming environments.

Appropriate Assessments for Measuring NE:
What type of assessment would be most appropriate for measuring Notational Expertise?

A study in Chapter 7 compared and evaluated four candidate assessment types for measuring NE that were created in response to deficiencies identified in previous assessments, and calls for the assessment of novice programming environments to be more robust. Findings from the study showed that results from Adapted Parsons Problems (APPs) had the highest correlation with results from code-writing questions, which are widely used to assess programming knowledge, and achieved a significantly higher response rate. Further studies using integrated APPs (presented in Chapter 8) confirmed that they were easy to use and achieved high response rates as formative assessment tools.

APPs also provide a novel way to assess student understanding of syntax structure in text, while also supporting the use of distractors, and providing the assessor with information about each student's current level of understanding, which can be used to adapt the current system or teaching style. APPs can also be easily extended in the future to assess structuring of visual elements to assess students' understanding of visual representations.

Inherent difficulties encountered when designing multiple-choice questions (MCQs) and debugging questions, and limits in the amount of information they are able to collect regarding students' (mis)conceptions of syntax structure, led to the conclusion that they were unlikely to provide useful measures of NE in isolation, and should instead be used as complementary questions to support summative assessments.

Motivation Intervention in MERs:
To what extent can motivation intervention features increase the use of symbolic representations and therefore improve acquisition of Notational Expertise?

This research question highlights a general concern relating to educational programming environments with one or more representations – how can the motivation for programming be provided, but also limited such that students achieve the maximum amount of learning without becoming distracted? This concern was highlighted in Cockburn and Bryant's investigation of Leogo (Cockburn & Bryant, 1997), and is arguably applicable to the majority of successful, modern educational programming tools that contain motivational features such as games, animations, and website creation. Findings presented in Chapter 8 strongly suggest that motivation intervention (MI) features increase students' acquisition of NE. However, improvements may have been due to factors other than an increase in the use of symbolic representations, such as an increase in reflection due to the limited number of interactions available. The research question can be further split into two parts:

The first part was concerned with whether MI features can increase the use of symbolic representations. Findings suggest that MI features can help to encourage students to use symbolic representations. Students using MI features spent more time viewing symbolic representations than the equivalent standard groups in all lessons except one, and in both studies. Groups using MI features improved significantly more than groups with standard features in the Lesson 1 of the first study of the chapter.

The second part was concerned with whether an increased use of symbolic representations leads to an increase in students' acquisition of NE. Findings show significant positive correlation between the length of time participants' spent viewing symbolic representations, and their total improvement, suggesting that viewing symbolic representations can help to improve acquisition of NE. Findings also showed that, on average, groups with MI features improved more than those with standard features over both lessons, and groups with MI features significantly improved in the second lesson, attaining the highest marks of all groups.

However, acquisition of NE improved for students using MI features who did not spend more time using symbolic representations, suggesting that motivation intervention features may have provided other benefits. One student stated in the post-study questionnaire "You have to think before you waste your coins". It is possible that the addition of motivation intervention features reduced unproductive tinkering, and increased reflection, thereby increasing acquisition of NE.

These collective findings suggest that MI features provide a useful mechanism with which to regulate students' use of motivation features within MER educational programming systems, and have the potential to increase the use of "hard" representations which are less appealing, but equally if not more important to students' acquisition of NE. Results also suggest that instead of unproductively tinkering, these types of features provide a "useful awkwardness" (Blackwell et al., 2001), which helps students to reflect on what they are doing and thereby increase their effectiveness when using the systems.

Year group-related Differences in MERs:
To what extent does student year group affect the acquisition of NE in DrawBridge or the impact of the motivation intervention?

It is important to consider whether the year group of the student has any effect on students' acquisition of NE in MER environments. Findings presented in Chapter 8 compared students from Year 8 and Year 9, and showed that students from higher year groups (Year 9) improved significantly more than students from lower year groups (Year 8). Findings also showed there was a significant interaction between the year group of the student and the inclusion of MI features on students' acquisition of NE, which suggested that acquisition of NE increases when older students are given MI features. Students from lower year groups using a version of DrawBridge with MI features improved the least in the first lesson of each study, suggesting that students from lower year groups took longer to gain benefit from the system when MI features were added.

This result is relevant to several stakeholders concerned with the development and use of educational programming tools with one or more representations. First, developers of future tools should be aware of the limitations of younger students – particularly when faced with a tool of reasonable complexity. Systems aimed at younger students should be adapted to meet their high usability requirements and help to support their (lack of) existing knowledge. Second, computing teachers should note that tools with high levels of complexity are unlikely to be appropriate for younger students, and may require more use to achieve the same results. Finally, educational bodies and policy makers should adapt their guidelines to reflect the realistic capabilities of younger students.

9.2 Contributions

The MoRA Framework

Researchers in Computer Science Education often build prototype systems to enable the study of new kinds of features or learning styles, of which many are still to be explored (Guzdial, 2004). In conventional education environments, the choice of representation affects the whole programming experience. Researchers therefore require guidance when choosing appropriate programming representations so that they can extend existing student knowledge, and help to create new knowledge that can be readily transferred in the future. The MoRA framework, presented in Chapter 3, provides a classification mechanism that can support such judgements, allowing researchers to classify types of representations according to their **mode** (tangible, graphical, symbolic or mental operation), and facility for **abstraction** (minimal, limited, and unlimited abstraction representations systems).

The MoRA framework can also be used to support the development of new MER-based educational programming environments, by helping to classify representations, identify relationships between representations, and identify strategies to make the transition from one representation to another. The framework also provides a stable foundation on which the research community can investigate different representation transitions, and more closely analyse the barriers between particular types of representation transition.

The MoRA framework was developed by drawing analogies between representation use in mathematics and programming education. Although it has been used in this thesis to develop strategies for the use of MERs in educational programming environments, the framework is likely to be applicable to other domains that also use many representations. For example, Physics education, in which the Investigative Science Learning Environment (ISLE) curriculum recommends the use of multiple representations when solving physics problems (Etkina & Van Heuvelen, 2001). These domains may also benefit in using a system like DrawBridge, in which representations were explicitly ordered with supported transitions.

DrawBridge: A Prototype MER Educational Programming Environment

To investigate transition strategies identified using the MoRA framework, a prototype MER environment, DrawBridge, was created (see Chapter 5). The environment used a theoretical basis to demonstrate how MERs can be used to effectively guide and support the transition between programming representations for students aged 7-14. Evidence presented in Chapter 6 and Chapter 8 indicated that students showed significant differences in improvement of Notational Expertise (NE) in as little as one double lesson. Results from Chapter 6 also suggest that the novel style of motivation (automatic segmentation of images drawn on paper to be used as objects to use in Programming by Demonstration (PbD)) provided major motivation to students.

DrawBridge is open source, and was developed to be flexible in order that different representation strategies could be explored. This flexibility allows representations to be placed in any order, at any position on the screen, and with any number of representations on the screen at once. DrawBridge also contains integrated Adapted Parson Problems, which, due to their modular nature, could easily be used independently of the system itself.

Novel Assessment Technique

Adapted Parsons Problems (APPs) are a method of assessment for individual statements or blocks of code. They were first introduced in Chapter 7 as a candidate assessment type for measuring Notational Expertise (NE), and extend traditional Parsons Problems (Parsons & Haden, 2006), which provide fully formed lines of code that users can drag and drop into position to form a complete program. Instead of lines of code, APPs provide individual syntax elements (e.g. semi-colons, brackets, keywords) that can be positioned to form a complete statement or block of code.

In this thesis, APPs were initially developed as paper-based assessments, and subsequently integrated into the DrawBridge tool as an integrated digital assessment. In addition to being an engaging, interactive type of assessment, APPs provide the assessor with valuable insight into the mental models of students who choose incorrect answers, and can therefore be used formatively by both researchers and teachers to adapt or direct future tools and teaching.

9.3 Limitations

Studies presented in this thesis suffered from a number of limitations due to several factors. Each limitation is addressed in its respective chapter, and summarised here to provide context for future work.

Limited Number of Lessons

Both sets of quasi-experiments reported in Chapter 6 and Chapter 8 took place over two double lessons in three different schools. Additional lessons would have provided understanding of students' use of MER systems over prolonged periods of time. The limit in the number of lessons was primarily due to the logistical challenges of setting up instrumentation, worksheets and lesson plans for four different groups for a prolonged period of time, and apprehension from school management that any longer would affect the use of pre-defined schemes of work. The studies could have been carried out in other contexts, such as an after school club or summer school for more prolonged periods of time. However, such studies would have been prone to limited external validity compared with real-world classroom scenarios due to confounding factors such as high ability as a result of student self-selection, mixed ages and experience, lack of focus and absence of teaching support.

Limitations of the DrawBridge System

The development of DrawBridge was a considerable amount of work (approximately 18,000 lines of Java) that took many months to complete and refine due to complexities related to the liveness and bi-directional nature of representations in the system. Due to these complexities, users reported that the system became sluggish on low-performance computers, which are common in many schools. Additionally, some DrawBridge components required recent features of the Java Runtime Environment, which had to be installed on PCs in all three schools used for the studies. Although on the whole DrawBridge worked well, more optimization work and bug fixing could have improved the usability of the system for some students.

Limited Initial Assessment Methodology

A two-part assessment was used to measure students' acquisition of NE in the first study, reported in Chapter 6. The assessment was found to have several limitations, such as low response rates, and the potential for students to infer answers from previous questions, which limited the validity of absolute assessment score comparisons. Pre-/Post-test gain score analysis was used to increase validity, and subsequent assessments (APPs) were rigorously evaluated to ensure high response rates and no ability to infer answers from previous questions.

9.4 Suggestions for Future Work

New Motivations for MERs

The DrawBridge MER system used characters drawn on paper as a starting representation for programming. The aim was to give students ownership over their program and encourage them to use programming to achieve their goals. Feedback from students during both sets of studies suggests that the characters provided a great source of motivation. However, there were limitations to this approach (e.g. the time taken to scan the image and the possibility of recognition failure), and it was not motivating to every student. It would therefore be worthwhile to investigate new sources of motivation that could be used to encourage students using MER systems. One alternate form of motivation might be to use physical robots as a starting representation. Students could build and directly manipulate a robot to create scaffolding code for programming, similar to the Picode system (Kato, Sakamoto, & Igarashi, 2013), and when they want to execute their programs, they could do so using the physical robot. A second type of motivation might be to use e-textiles, such as the LilyPad Arduino Kit, which could provide similar benefits, and may be particularly compelling for girls (Buechley, Eisenberg, & Catchen, 2008).

MER Benefits for Older Students

The final studies reported in this thesis, described in Chapter 8, compared groups of students in Year 8 (aged 12-13) and Year 9 (aged 13-14) to measure the difference in acquisition of NE. The findings showed that students from higher year groups improved significantly more than students from lower year groups, suggesting that older students benefit more from using the system. Future work might therefore investigate a wider difference in age to continue developing recommendations for teachers and researchers using MERs for educational programming.

Older students at undergraduate level are regularly taught with programming environments such as Alice, BlueJ and Scratch at both CS0 level (preparatory course in the USA) and CS1 (introductory Computer Science) (Dann et al., 2012; Jadud, 2005; Mishra, 2014). These environments are used to scaffold learning of text programming languages, in the same way that teachers scaffold learning in schools (see Chapter 4). The similarities between representation-transition strategies in these courses, and in schools, in addition to the increased improvement of students in higher year groups suggests that the use of MERs with undergraduate students would be a promising future research direction. Furthermore, undergraduate students are expected to make the transition more quickly than students in school, and have a stable, mature level of cognitive development, which suggests that a single MER system might provide even more benefit.

Alternative Semantics and Paradigms

Studies presented in this thesis were carried out using DrawBridge, a prototype MER system that supported representations rooted in the semantics of JavaScript. The modification of DrawBridge to support alternative semantics would have been a significant amount of work, and was therefore not possible in the time available. Future studies into the use of MERs in educational programming environments should investigate the use of alternate semantics or programming paradigms to verify that the results of this thesis apply to different semantics, and investigate whether MERs might help students when making the transition between different languages or paradigms.

Comparison with Code Morphing Environments

Recent commercial educational development environments, such as the Code Kingdoms system used in Chapter 7, have used “code morphing” mechanisms to allow students to make the transition between programming representations. Code morphing allows students to click a button or drag a slider to change the appearance of programming code without changing the position of each element. The change in representation uses a “hugging” method of transfer (Perkins & Martin, 1986), which ensures that the context and actions that can be done on the elements are as similar as possible. Students may gain benefits from code morphing insomuch as they can move to the new representation, but may also be disadvantaged, as the mechanism may reduce their need, and ability, to translate between representations. Further theoretical and empirical comparisons should be made to investigate the differences.

The Need for Notational Expertise in the Future

Many researchers argue that as technology becomes more ubiquitous, humans will be required to learn some kind of formal notation in order to interact with computers. For example, Norman argues that Natural Language User Interfaces (NLUI) are not natural, and that they contain a set of possible actions, and methods to evoke each action, which the user must learn in order to use the system (Norman, 2010). Blackwell and Green use the Cognitive Dimensions (CDs) of Notations framework to highlight deficiencies in voice interaction, such as the restriction in order of words (high *premature commitment* $^{(CD)}$), and difficulty in changing parts of the notation (high *viscosity* $^{(CD)}$), which demonstrate the formal nature of the interaction mechanism (Blackwell & Green, 2003). I argue that as gestural and voice interaction become more common, non-technical users will need to acquire notational expertise skills. MER systems could provide a basis on which to develop tools to assist users with this goal.

9.5 Closing Remarks

This thesis has demonstrated that Multiple External Representations (MERs) can be used to improve student acquisition of Notational Expertise (NE) – a set of mental competencies that can be used to master the use of one or more notations, identify ideas embedded within notations, and translate from one notation to another (see Chapter 3 for a full definition).

Studies conducted using a prototype MER educational programming system (DrawBridge) validated representation transition strategies identified using the Modes of Representational Abstraction (MoRA) framework, a novel framework that enables the classification and development of strategies to make transitions between representations, presented in Chapter 3. Results from studies also provided evidence to support the requirement for low-abstract representations to provide motivation for students.

A robust, novel assessment type, Adapted Parsons Problems (APP), was developed and evaluated to measure Notational Expertise. The assessment was successfully used to measure the impact of motivation intervention features, added to DrawBridge to guide students to make the transition towards abstract symbolic representations. Results suggested that these features were successful, and that they may also have encouraged student reflection by limiting the number of interactions they could make. As a result, this thesis has provided empirical, theoretical and practical evidence for the use of multiple representations within educational programming, thus forming a basis on which MERs can be developed to be used in schools, and a platform for future research.

Bibliography

Aaron, S., & Blackwell, A. F. (2013). From Sonic Pi to Overtone : Creative Musical Experiences with Domain-Specific and Functional Languages. *The First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, 35–46.

ACM. (2010). *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age*.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, *9*(4), 422–433.

Adler, I., & Adler, R. (1959). *The New Mathematics*. Dobson.

Ainsworth, S. (1997). *Designing and Evaluating Multi-Representational Environments*. PhD thesis. University of Nottingham.

Ainsworth, S. (1999). The Functions of Multiple Representations. *Computers & Education*, *33*, 131–152.

Ainsworth, S. (2006). DeFT : A conceptual framework for considering learning with multiple representations, *16*, 183–198.

Ainsworth, S., & Van Labeke, N. (2002). Using a Multi-Representational Design Framework to Develop and Evaluate a Dynamic Simulation Environment. *International Workshop on Dynamic Visualizations and Learning*.

Anghileri, J., & Beishuizen, M. (1998). Counting Chunking and the Division Algorithm. *Mathematics in School*, *27*(1), 2–4.

Baecker, R. (1988). Enhancing program readability and comprehensibility with tools for program visualization. *Proceedings of 11th International Conference on Software Engineering*, 356–366.

Bandura, A. (1977). Self-efficacy: Toward a Unifying theory or Behavioral change. *Psychological Review*, *82*(2), 191.

Bandura, A. (1997). *Self-efficacy: The Exercise of Control* (Vol. 72).

Barnett, S. M., & Ceci, S. J. (2002). When and where do we apply what we learn? A taxonomy for far transfer. *Psychological Bulletin*, *128*(4), 612–637.

Bates, E. (1979). *The Emergence of Symbols: Cognition and Communication in Infancy*. Academic Press.

Bauer, J. (2005). Toward Technology Integration in the Schools : Why It Isn ' t Happening. *Journal of Technology and Teacher Education*, *13*, 519–546.

BBC. (2011). Google's Eric Schmidt criticises education in the UK. Retrieved January 30, 2015, from http://www.bbc.co.uk/news/uk-14683133

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students. *ACM SIGCSE Bulletin*, *37*(2), 103–106.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., & Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '06*, 231–240.

Beckwith, L., Sorte, S., Burnett, M., Wiedenbeck, S., Chintakovid, T., & Cook, C. (2005). Designing Features for Both Genders in End-User Programming Environments. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 153–160.

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2008). Computer science without computers:new outreach methods from old tricks, 127–133.

Bingimlas, K. A. (2009). Barriers to the Successful Integration of ICT in Teaching and Learning Environments: A Review of the Literature. *Eurasia Journal of Mathematics, Science & Technology Education*, *5*(3), 235–245.

Black, P., & Wiliam, D. (1998). *Assessment and Classroom Learning. Assessment in Education: Principles, Policy & Practice* (Vol. 5).

Blackwell, A. F. (2002a). First steps in programming: a rationale for attention investment models. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10.

Blackwell, A. F. (2002b). What is programming? In *14th workshop of the Psychology of Programming Interest Group* (pp. 204–218).

Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., … Young, R. M. (2001). Cognitive Dimensions of Notations: Design Tools for Cognitive Technology., 325–341.

Blackwell, A. F., & Engelhardt, Y. (1998). A Taxonomy of Diagram Taxonomies. In *Proceedings of Thinking with Diagrams 98: Is there a science of diagrams* (pp. 60–70).

Blackwell, A. F., & Green, T. R. G. (2003). Notational Systems – the Cognitive Dimensions of Notations framework. In J. M. Carroll (Ed.), *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science* (pp. 1–21). Morgan Kaufmann.

Bossé, M. J., Adu-Gyamfi, K., & Cheetham, M. (2011). Transitions Among Mathematical Representatins: Teachers Beliefs and Practices. *International Journal of Mathematics Teaching and Learning*, *15*(6), 1–23.

Bradley, D., & Roth, G. (2007). Adaptive Thresholding using the Integral Image. *Journal of Graphics, GPU, and Game Tools*, *12*, 13–21.

Brown, N., & Altadmri, A. (2014). Investigating novice programming mistakes: educator beliefs vs. student data. In *International computing education Research* (pp. 43–50).

Bruner, J. S. (1960). *The Process of Education*. Harvard University Press.

Bruner, J. S. (1968). *Toward a Theory of Instruction. Harvard University Press*.

Buechley, L., Eisenberg, M., & Catchen, J. (2008). The LilyPad Arduino: Using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. *CHI '08 Proceedings of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems*, 423–432.

Busch, T. (1995). Gender Differences in Self-Efficacy and Attitudes Toward Computers. *Journal of Educational Computing Research*, *12*, 147–158.

Campbell, D. T., & Stanley, J. C. (1967). *Experimental and Quasi-Experimental Design for Research*. *Handbook of Research on Teaching (1963)*. doi:10.1037/022808

Cockburn, a., & Bryant, a. (1997). Leogo: An Equal Opportunity User Interface for Programming. *Journal of Visual Languages & Computing*, *8*(5-6), 601–619. doi:10.1006/jvlc.1997.0152

Code Kingdoms. (2014). Code Kingdoms. Retrieved from www.codekingdoms.com

Code.org. (2015). K-5 Curriculum Overview. Retrieved April 11, 2014, from http://code.org/educate/k5

CodeClub. (2015). CodeClub. Retrieved April 11, 2015, from https://www.codeclub.org.uk

Collins, A., Brown, J. S. S., & Newman, S. E. (1989). Cognition apprenticeship: reaching the craft of reading, writing, and mathematics. In *Knowing, learning, and instruction : Essays in honor of Robert Glaser* (pp. 32–42). Lawrence Erlbaum Associates.

Collins, H., & Target, R. (2013). CodeKingdoms. Retrieved March 9, 2015, from www.codekingdoms.com

Cox, R. (1996). Analytical reasoning with multiple external representations.

Cox, R., & Brna, P. (1995). Supporting the use of external representations in problem solving: The need for flexible learning environments. *Journal of Artificial Intelligence in Education*, *6*(November), 239–302.

Cutts, Q., Esper, S., Fecho, M., Foster, S. R., & Simon, B. (2012). The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition (pp. 63–70).

Cypher, A. (1991). EAGER: Programming Repetative Tasks By Example. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 33–39).

Cypher, A. (1993). *Watch What I Do: Programming By Demonstration*. MIT Press.

Dann, W., Cosgrove, D., Slater, D., & Culyba, D. (2012). Mediated Transfer : Alice 3 to Java. In *SIGCSE '12 Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 141–146). Raleigh, North Carolina, USA: ACM.

Dehnadi, S., & Bornat, R. (2006). The Camel has Two Humps. In *Little PPIG* (pp. 1–21).

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *ICER '08 Proceedings of the Fourth international Workshop on Computing Education Research* (pp. 113–124). Sydney, Australia.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the Syntax Barrier for Novices. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education - ITiCSE '11*, 208–212.

DfE. (2013). *Computing programmes of study : Key Stages 3 and 4*.

Dimitrov, D. M., & Rumrill, P. D. (2003). Pretest-posttest designs and measurement of change. *Work (Reading, Mass.)*, *20*, 159–165.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Eich, B. (1996). JavaScript 1.1 Specification. Retrieved March 4, 2015, from http://hepunx.rl.ac.uk/~adye/jsspec11/jsrefspe.htm

Engelhardt, Y. (2002). *The Language of Graphics. A framework for the analysis of syntax and meaning in maps, charts and diagrams*. PhD Thesis. University of Amsterdam.

Ericson, B., Guzdial, M., & Mcklin, T. (2014). Preparing Secondary Computer Science Teachers Through an Iterative Development Process, 116–119.

Ericson, B., & McKlin, T. (2012). Effective and sustainable computing summer camps. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE '12*, 289.

Esper, S., Foster, S. R., & Griswold, W. G. (2013). On the nature of fires and how to spark them when you're not there. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, 305.

Esper, S., Foster, S. R., Griswold, W. G., Herrera, C., & Snyder, W. (2014). CodeSpells: Bridging Educational Language Features with Industry-Standard Languages. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (pp. 5–14). ACM.

Esper, S., Wood, S. R., Foster, S. R., Lerner, S., & Griswold, W. G. (2014). Codespells: how to design quests to teach java concepts. *Journal of Computing Sciences in Colleges*, *29*, 114–122.

Etkina, E., & Van Heuvelen, A. (2001). Investigative Science Learning Environment: Using the processes of science and cognitive strategies to learn physics. In *Proceedings of the 2001 Physics Education Research Conference* (pp. 17–20).

Feynman, R. P. (1965). New textbooks for the “new” mathematics. *Engineering and Science*, *28*(March), 9–15.

Fincher, S. (1999). What are We Doing When We Teach Programming? *Frontiers in Education Conference*, *1*, 8–12.

Flannery, L. P., Ave, C., Kazakoff, E. R., Silverman, B., Bers, M. U., & 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). ACM.

Fowler, M. (2008). Projectional Editing. Retrieved September 13, 2013, from http://martinfowler.com/bliki/ProjectionalEditing.html

Franklin, D., Kiefer, B., Laird, C., Lopez, F., Pham, C., Suarez, J., … Almeida-Tanaka, P. (2013). Assessment of Computer Science Learning in a Scratch-Based Outreach Program. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE ’13*, 371. doi:10.1145/2445196.2445304

Frei, P., Su, V., Mikhak, B., & Ishii, H. (2000). curlybot : Designing a New Class of Computational Toys. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 129–136).

Gagatsis, A., & Shiakalli, M. (2004). Ability to Translate from One Representation of the Concept of Function to Another and Mathematical Problem Solving. *Educational Psychology*, *24*(5), 645–657.

Gibbs, G. (2007). *Analysing Qualitative Data - The Sage Qualitative Research Kit. The SAGE Qualitative Reseach Kit*. SAGE.

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*(1), 31–48.

Gluga, R., Kay, J., Lister, R., & Teague, D. (2012). On the reliability of classifying programming tasks using a neo-piagetian theory of cognitive development. *Proceedings of the Ninth Annual International Conference on International Computing Education Research - ICER '12*, 31. doi:10.1145/2361276.2361284

Google. (2013). What is Computational Thinking. Retrieved January 2, 2013, from http://www.google.com/edu/computational-thinking/what-is-ct.html

Google. (2015). CS-First. Retrieved April 11, 2015, from http://www.cs-first.com/overview

Green, T. R. G., & Blackwell, A. F. (1998). Cognitive Dimensions of Information Artefacts : a tutorial. *Applied Psychology*, (October).

Green, T. R. G., & Petre, M. (1992). When visual programs are harder to read than textual programs. *... Interaction: Tasks and Organisation, Proceedings of ...*, (1987).

Green, T. R. G., & Petre, M. (1996a). Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. In *Journal of Visual Languages and Computing* (pp. 1–51).

Green, T. R. G., & Petre, M. (1996b). Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages & Computing*, (January), 1–51.

Gross, P., & Powers, K. (2005). Evaluating assessments of novice programming environments. *Proceedings of the 2005 International Workshop on Computing Education Research - ICER '05*, 99–110. doi:10.1145/1089786.1089796

Gulz, A. (2004). Benefits of Virtual Characters in Computer Based Learning Environments : Claims and Evidence. *Internation Journal of Articial Intelligence in Education*, *14*, 313–334.

Guzdial, M. (2003). A media computation course for non-majors. *ACM SIGCSE Bulletin*, *35*(3), 104–108.

Guzdial, M. (2004). Programming Environments for Novices. In *Computer science education research* (pp. 127–154). Taylor & Francis, Abingdon, UK.

Haake, M. (2009). *Embodied Pedagogical Agents*. Lund University.

Haeberli, P. (1988). ConMan: a visual programming language for interactive graphics. *ACM SigGraph Computer Graphics*, *22*(4), 1–51.

Halsey, a. H., & Sylva, K. (1987). Plowden: history and prospect. *Oxford Review of Education*, *13*(February 2015), 3–11. doi:10.1080/0305498870130101

Hamari, J., Koivisto, J., & Sarsa, H. (2014). Does Gamification Work?--A Literature Review of Empirical Studies on Gamification. *System Sciences (HICSS), 2014 47th Hawaii International Conference on*, 3025–3034.

Hepburn, G., & Buley, J. (2006). Getting open source software into schools: Strategies and challenges. *Innovate. Journal of Online Education*.

Heritage, M., & Niemi, D. (2011). Toward a Framework for Using Student Mathematical Representations as Formative Assessments. *Educational Assessment*, (May 2013), 37–41.

Higher Education Funding Council for England (HEFCE). (2013). *Non-continuation rates at English HEIs: Trends for entrants 2005-06 to 2010-11.*

Hundhausen, C., & Brown, J. (2007). What You See Is What You Code: A “live” algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, *18*(1), 22–47. doi:10.1016/j.jvlc.2006.03.002

Hundhausen, C., Farley, S., & Brown, J. L. (2006). Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An

Experimental Study. *Visual Languages and Human-Centric Computing (VL/HCC'06)*, 157–164. doi:10.1109/VLHCC.2006.12

Huotari, K. (2012). Defining Gamification - A Service Marketing Perspective. *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, 17–22. doi:10.1145/2393132.2393137

Ishii, H., & Ullmer, B. (1997). Tangible bits: towards seamless interfaces between people, bits and atoms. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 241.

ISTE. (2013). Computational Thinking Vocabulary and Progression Chart. *The International Society for Technology in Education*. Retrieved January 5, 2013, from http://www.iste.org/docs/ct-documents/ct-teacher-resources_2ed-pdf.pdf?sfvrsn=2

Jadud, M. C. (2005). A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, *15*(1), 25–40. doi:10.1080/08993400500056530

Joseph, C., & Siganakis, T. (2014). Which Language Wins in Terms of Salary/Demand. Retrieved March 10, 2015, from https://gooroo.io/GoorooTHINK/Article/16191/Which-language-wins-in-terms-of-salarydemand-July-2014/14105#.VP9Z_kIkKS0

Kato, J., Sakamoto, D., & Igarashi, T. (2013). Picode: Inline Photos Representing Posture Data in Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3097–3100). Paris, France: ACM.

Kay, A. (1972). A Personal Computer for Children of All Ages. *Proceedings of the ACM National Conference*, (I), 1. doi:10.1145/800193.1971922

Kay, A. (1991). User Interface: A Personal View. In B. Laurel (Ed.), *The Art of Human-Computer Interface Design* (pp. 191–207). Addison-Wesley.

Kelleher, C. (2006). *Motivating Programming : using storytelling to make computer programming attractive to middle school girls*. PhD Thesis. Carnegie Mellon University.

Kelleher, C., Cosgrove, D., & Culyba, D. (2002). Alice2: programming without syntax errors. In *User Interface Software ...* (pp. 3–4).

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), 83–137.

King, N., Cassell, C., & Symon, G. (2004). Using Templates in the Thematic Analysis of Texts. *Essential Guide to Qualitative Methods in Organizational Research*, 256–270. doi:Book

Kinnunen, P., & Malmi, L. (2007). Why Students Drop Out CS1 Course? *Educational Leadership*, *64*, 91.

Kirschner, P. A. (2002). Cognitive load theory: implications of cognitive load theory on the design of learning. *Learning and Instruction*, *12*(1990), 1–10. doi:10.1016/S0959-4752(01)00014-7

Kline, M. (1973). *Why Johnny Can't Add: The Failure of the New Math*. New York: St. Martin's Press.

Knuth, E. J., Stephens, A. C., McNeil, N. M., & Alibali, M. W. (2006). Does Understanding the Equal Sign Matter? Evidence from Solving Equations. *Journal for Research in Mathematics Education*, *37*(4), 297–312. doi:10.2307/30034852

Ko, A. J., Blackwell, A. F., Burnett, M., Erwig, M., Lawrance, J., Lieberman, H., … Rosson, M. B. (2011). The State of the Art in End-User Software Engineering. *ACM Computing Surveys (CSUR*, *43*(3).

Ko, A. J., & Myers, B. (2005). Citrus : A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data. *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, 3–12.

Ko, A. J., Myers, B. A., Rosson, M. B., Rothermel, G., Shaw, M., Wiedenbeck, S., … Lieberman, H. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys*, *43*(3), 1–44.

Koivisto, J., & Hamari, J. (2014). Computers in Human Behavior Demographic differences in perceived benefits from gamification, *35*, 179–188.

Kolling, M. (2010). The Greenfoot Programming Environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1–21. doi:10.1145/1868358.1868361.http

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy, *13*(4), 1–12. doi:10.1076/csed.13.4.249.17496

Krathwohl, D. (2002). A Revision of Bloom's Taxonomy : An Overview. In *Theory into practice* (pp. 37–41). doi:10.1207/s15430421tip4104

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *ITiCSE '05 Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 14–18). New York, New York, USA: ACM Press. doi:10.1145/1067445.1067453

Lampert, M. (1986). Knowing, doing, and teaching multiplication. *Cognition and Instruction*, *3*(4), 305–342.

Larkin, J., & Simon, H. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, *99*, 65–99. doi:10.1016/s0364-0213(87)80026-5

Lee, M. J., & Ko, A. J. (2012). Investigating the role of purposeful goals on novices' engagement in a programming game. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* (pp. 163–166). doi:10.1109/VLHCC.2012.6344507

Lee, M. J., Ko, A. J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research - ICER '13*, 153. doi:10.1145/2493394.2493410

Lesh, R. (1999). The development of representational abilities in middle school mathematics. In *I. E. Sigel (Ed.), Development of Mental Representation: Theories and Application* (pp. 323–350). Hillsdale, NJ: Lawrence Erlbaum Associates.

Levy, R. B.-B., Ben-Ari, M., & Uronen, P. (2003). The Jeliot 2000 Program Animation System. *Computers & Education*, *40*(1), 1–15. doi:10.1016/S0360-1315(02)00076-3

Lewis, C., Esper, S., & Bhattacharyya, V. (2014). Children's Perception of What Counts as a Programming Language. *Journal of Computing Education Research*, 123–133.

Lieberman, D. (1999). *Learning: Behavior and Cognition*. Wadsworth Publishing Co Inc.

Lin, J., Newman, M., Hong, J., & Landay, J. (2001). DENIM: An Informal Sketch-based Tool for Early Stage Web Design. *Video Poster in Extended Abstracts of Human Factors in Computer Systems: CHI*, (Figure 1), 205–206.

Linn, R. (1995). *Measurement and Assessment in Teaching*. Pearson Education.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., … Thomas, L. (2004). A Multi-national Study of Reading and Tracing Skills in Novice Programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150.

Lockton, D. (2015). Design with Intent Toolkit. Retrieved April 20, 2015, from http://designwithintent.co.uk

Lockton, D., Harrison, D., & Stanton, N. (2008). Design with intent: Persuasive technology in a wider context. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *5033 LNCS*, 274–278. doi:10.1007/978-3-540-68504-3-30

Mackinnon, S. P., Jordan, C. H., & Wilson, A. E. (2011). Birds of a feather sit together: physical similarity predicts seating choice. *Personality and Social Psychology Bulletin*, *37*(7), 879–892. doi:10.1177/0146167211402094

MacLaurin, M. B. (2011). The design of kodu: a tiny visual programming language for children on the Xbox 360. *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '11*, 241. doi:10.1145/1926385.1926413

Major, L., Kyriacou, T., & Brereton, P. (2011). Simulated Robotic Agents as Tools to Teach Introductary Programming. *Computing*, (March), 3837–3846.

Maleki, M. M. (2013). Programming in the Model: A New Scripting Interface for Parametric CAD Systems. *Association for Computer-Aided Design In Architecture*, 191–198.

Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2008). Programming by Choice : Urban Youth Learning Programming with Scratch. In *ACM SIGCSE Bulletin* (Vol. 40, pp. 367–371). ACM.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 16:1–16:15. doi:10.1145/1868358.1868363

Margolis, J., & Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing*. MIT Press.

May, H. (2012). Nonequivalent Comparison Group Designs. In *APA Handbook of Research Methods in Psychology, Vol 2: Research designs: Quantitative, Qualitative, Neuropsychological, and Biological* (Vol. 2, pp. 489–509). Washington, DC, US: American Psychological Association. doi:10.1037/13620-026

Mayer, R. E., & Sims, V. K. (1994). For whom is a picture worth a thousand words? Extensions of a dual-coding theory of multimedia learning. *Journal of Educational Psychology*, *86*(3), 389–401. doi:10.1037/0022-0663.86.3.389

McCarty, J., & Shrum, L. (2000). The Measurement of Personal Values in Survey Research: A Test of Alternative Rating Procedures. *Public Opinion Quarterly*.

McGill, M. M. (2012). Learning to Program with Personal Robots. *ACM Transactions on Computing Education*, *12*(1), 1–32. doi:10.1145/2133797.2133801

McIver, L. (2000). The Effect of Programming Language on Error Rates of Novice Programmers. In *Proceedings of the Psychology of Programming* (pp. 181–192).

McIver, L., & Conway, D. (1996). Seven deadly sins of introductory programming language design. *Software Engineering: Education and …*.

McKay, F. (2012). A prototype structured but low-viscosity editor for novice programmers. In *eWic HCI*.

Meerbaum-Salant, O. (2010). Learning computer science concepts with scratch. *ICER '10: Proceedings of the Sixth International Workshop on Computing Education Research*, 69–76. doi:10.1145/1839594.1839607

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in scratch. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education - ITiCSE '11*, 168–172.

Microsoft. (2008). Microsoft Small Basic. Retrieved January 30, 2015, from http://smallbasic.com

Mishra, S. (2014). Effect of a 2-week Scratch Intervention in CS1 on Learners with Varying Prior Knowledge. *Proceedings of the 19th ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '14*, 45–50.

Morgado, L., & Kahn, K. (2008). Towards a specification of the ToonTalk language. *Journal of Visual Languages & Computing*, *19*(5), 574–597. doi:10.1016/j.jvlc.2007.10.002

Morrison, B. B., & DiSalvo, B. (2014). Khan academy gamifies computer science. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education - SIGCSE '14*, 39–44. doi:10.1145/2538862.2538946

Mulholland, P., & Watt, S. (1998). Hank : A Friendly Cognitive Modelling Language for Psychology Students, (September), 1–4.

Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to "explain in plain english" linked to proficiency in computer-based programming. *Proceedings of the Ninth Annual International Conference on International Computing Education Research - ICER '12*, 111. doi:10.1145/2361276.2361299

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123. doi:10.1016/S1045-926X(05)80036-9

Nash, C. (2012). Supporting virtuosity and flow in computer music.

Nathan, M. J., Stephens, A. C., Masarik, K., Alibali, M. W., & Koedinger, K. R. (2002). Representational fluency in middle school: A classroom study. In *Proceedings of the Twenty-Fourth Annual Meeting of the North American Chapter of the International Group for the Psychology of Mathematics Education, Vol. 1* (pp. 462–474).

National Centre for Education Statistics. (2013). *STEM Attrition: College Students' Paths Into and Out of STEM Fields*.

Nevins, C. (2009). *The Effect of Correspondence Highlighting on Novice Programmer Instruction*. Washington State University.

Niemi, D. (1996). Assessing in Mathematics : Problem and Conceptual Understanding Representations, Solutions, Justifications, and Explanations. *The Journal of Educational Research*, *89*(6), 351–363.

Norman, D. (1988). *The Design of Everyday Things*.

Norman, D. a. (2010). The way I see it: Natural user interfaces are not natural. *Interactions*, *17*(3), 6. doi:10.1145/1744161.1744163

North, C., & Shneiderman, B. (2000). Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *Proceedings of the working conference on Advanced visual interfaces*. doi:10.1145/345513.345282

Nunes, T., Schliemann, A. D., & Carraher, D. W. ; (1993). *Street Mathematics*. Cambridge University Press.

O'Grady, S. (2015). The RedMonk Programming Language Rankings: January 2015. *RedMonk*. Retrieved from http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

O'Hara, K. P., & Payne, S. J. (1999). Planning and the user interface: the effects of lockout time and error recovery cost. *International Journal of Human-Computer Studies*, *50*(1), 41–59. doi:10.1006/ijhc.1998.0234

Oracle. (n.d.). Learn About Java Technology. *2015*. Retrieved March 11, 2015, from https://www.java.com/en/about/

Pane, J. F., & Myers, B. A. (1996). *Usability Issues in the Design of Novice Programming Systems*.

Papert, S. (1980). Mindstorms: Children, Computers and Powerful Ideas. *New Ideas in Psychology*. New York, NY, USA: Basic Books, Inc. doi:10.1016/0732-118X(83)90034-X

Papert, S., & Turkle, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, *11*(1), 3–33.

Parnafes, O., & Disessa, A. (2004). Relations between types of reasoning and computational representations. *International Journal of Computers for Mathematical Learning*, *9*, 251–280. doi:10.1007/s10758-004-3794-7

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. *Proceedings of the 8th Australasian Conference on Computing Education*, *52*, 157–163. doi:10.1007/s13398-014-0173-7.2

Parsons, D., & Haden, P. (2007). Programming osmosis: Knowledge transfer from imperative to visual programming environments. *Conference of the National Advisory Committee on Computing Qualifications*, (Naccq), 209–215.

Perkins, D., & Martin, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In *First Workshop on Empirical Studies of Programmers* (pp. 213–229).

Perkins, D., & Salomon, G. (1988). Teaching for Transfer. *Educational Leadership*, *46*(1), 22–32.

Perlman, R. (1976). Using Computer Technology to Provide a Creative Learning Environment for Preschool Children. *MIT Logo Memo #24,*.

Piaget, J., & Inhelder, B. (1969). *The Psychology of the Child*. Basic Books.

Pimm, D. (2002). *Symbols and meanings in school mathematics*. Routledge.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with Alice. *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education*, *1*, 213–217. doi:10.1145/1227310.1227386

Resnick, M., Maloney, J., Andres, M.-H., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch : Programming for All. *Communications of the ACM*, *52*(11), 60–67.

Rigby, P. C., & Thompson, S. (2005). Study of novice programmers using Eclipse and Gild. *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '05*, 105–109. doi:10.1145/1117696.1117718

Ryan, A. M., Gheen, M. H., & Midgley, C. (1998). Why do some students avoid asking for help? An examination of the interplay among students' academic efficacy, teachers' social–emotional role, and the classroom goal structure. *Journal of Educational Psychology*, *90*(3), 528–535. doi:10.1037/0022-0663.90.3.528

Ryan, G. W., & Bernard, H. R. (2003). Techniques to Identify Themes. *Field Methods*, *15*(1), 85–109. doi:10.1177/1525822X02239569

Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, *2*(1), 51–72. doi:10.1080/10447319009525970

Sellen, A. J., & Harper, R. H. R. (2003). *The Myth of the Paperless Office*. Cambridge, MA, USA: MIT Press.

Shimojima, A. (1999). The Graphic-Linguistic Distinction Exploring Alternatives. *Artificial Intelligence Review*, *13*(4), 313–335.

Shneiderman, B. (1981). Direct Manipulation: a step beyond programming languages. In R. M. B. and W. A. S. Buxton (Ed.), *Human-Computer Interaction: A Multidisciplinary Approach* (Vol. 13, pp. 461–467). Los Altos, CA: Morgan Kaufmann Publishers.

Simon, B., Kohanfars, M., Lee, J., Tamayo, K., & Cutts, Q. (2010). Experience report: peer instruction in introductory computing. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 341–345). ACM.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., … Warburton, G. (2012). Introductory programming : examining the exams. In *Fourteength Australasian Computing Education Conference (ACE2012)* (pp. 61–70). Melbourne, Australia.

Sinclair, A. J. (2014). Educational Programming Languages: The Motivation to Learn with Sonic Pi. In *Psychology of Programming Interest Group* (pp. 215–228). Brighton, UK.

Smith, D. C., Cypher, A., & Tesler, L. (2000). Novice Programming Comes of Age. *Communications of the ACM*, *43*(3), 75–81.

Stenning, K., & Oberlander, J. (1995). A Cognitive Theory of Graphical and Linguistic Reasoning: Logic and Implementation. *Cognitive Science*, 97–140.

Suzuki, H., & Kato, H. (1995). Interaction-level Support for Collaborative Learning: AlgoBlock - An Open Programming Language. *The First International Conference on Computer Support for Collaborative Learning - CSCL '95*. doi:10.3115/222020.222828

Sweller, J., Chandler, P., Tierney, P., & Cooper, M. (1990). Cognitive load and selective attention as factors in the structur- ing of technical material. *Journal of Experimental Psychology: General*, *119*, 176–192. doi:10.1037/0096-3445.119.2.176

Tanimoto, S. (2013). A perspective on the evolution of live programming. *Workshop on Live Programming (LIVE)*.

Tanimoto, S. L. (1990). VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing*, *1*(2), 127–139.

Taub, R., Ben-Ari, M., & Armoni, M. (2009). The effect of CS unplugged on middle-school students' views of CS. *ACM SIGCSE Bulletin*, *41*, 99. doi:10.1145/1595496.1562912

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013). A Qualitative Think Aloud Study of Early Neo-Piagetian Stages of Reasoning in Novice Programmers, *136*.

Teague, D., & Lister, R. (2014). Blinded by their Plight : Tracing and the Preoperational Programmer. In *The Psychology of Programming Interest Group (PPIG)*.

The Independent ICT in Schools Commission. (1997). *Information and Communications Technology in UK Schools: An Independent Inquiry*. Financial Times.

The National Academy of Sciences. (2010). *Report of a Workshop on The Scope and Nature of Computational Thinking*. The National Academy for Sciences.

The National Academy of Sciences. (2011). *Report of a Workshop of Pedagogical Aspects of Computational Thinking Committee for the Workshops on Computational Thinking*.

The Royal Society. (2012). Shut down or restart? The way forward for computing in UK schools. *Technology*, (January), 1–122.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. *Conferences in Research and Practice in Information Technology Series*, *78*(January), 155–161.

Turkle, B. S., & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Culture*, *11*(1), 1–32.

Vihavainen, A., Paksula, M., Luukkainen, M., & Kurhila, J. (2011). Extreme Apprenticeship Method : Key Practices and Upward Scalability, 273–277.

Vygotsky, L. S. (1978). Mind in society. *Mind in Society The Development of Higher Psychological Processes*, *Mind in So*, 159. doi:10.1007/978-3-540-92784-6

Wagner, R. a., & Fischer, M. J. (1974). The String-to-String Correction Problem. *Journal of the ACM*, *21*(1), 168–173. doi:10.1145/321796.321811

Weaver, C. (2004). Building Highly-Coordinated Visualizations in Improvise. *IEEE Symposium on Information Visualization*, 159–166. doi:10.1109/INFVIS.2004.12

Wells, D. (2012). Computing in schools: time to move beyond ICT?, *2*(1), 8–13.

Williams, L., & Upchurch, R. L. (2001). In support of student pair-programming. *ACM SIGCSE Bulletin*, *33*, 327–331. doi:10.1145/366413.364614

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33.

Wing, J. M. (2008). Computational Thinking and Thinking about Computing. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences*, *366*(1881), 3717–3725.

Wing, J. M. (2014). Wheeler lecture: "Computational Thinking." 14th May 2014. Cambridge, UK.

Wood, D., Bruner, J. S., & Ross, G. (1976). The Role of Tutoring is Problem Solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100.

Woodford, K., & Bancroft, P. (2005). Multiple Choice Questions Not Considered Harmful. *ACE '05 Proceedings of the 7th Australasian Conference on Computing Education*, *42*, 109–116.

Zhang, J. (1997). The Nature of External Representations in Problem Solving. *Cognitive Science*, *21*(2), 179–217. doi:10.1207/s15516709cog2102_3

Zhang, J., & Patel, V. L. (2006). Distributed cognition, representation, and affordance. *Pragmatics & Cognition*, *14*, 333–341. doi:10.1075/pc.14.2.12zha

Zhengyou Zhang, L. H. (2002). *Whiteboard It ! Convert Whiteboard Content into an Electronic Document*.

Appendix A

Semi-Structured Teaching Interview Questions

Section 1: Current Practices

Are there any concepts that students struggle with between KS2 and KS3?

Are there concepts where understanding is particularly dependent on students' ability?

What are the current tools you use for

- Web development
- Text-based programming
- Pre-text programming

Thinking about visual languages

- Do you use it to teach? If so, how? If not, why not?
- Benefits/Problems?

Thinking about paper

- Do you use it to teach? If so, how? If not, why not?
- Benefits/Problems?

Have you heard of CSUnplugged? Do you use it? If not, why not?

Thinking about physical computing/tangibles

- Do you use them to teach? If so, how? If not, why not?
- Have you heard of LEGO Mindstorms?
- Have you used/considered using robots?
- Benefits/Problems?

Thinking about text-based languages

- Do you use them to teach? If so, how? If not, why not?
- Benefits/Problems?

Are there any problems in general with those tools?

Are there any tools you have been tempted to use but avoided?

Have you ever used any of these systems side-by-side? If so, for what reason?

How do you ensure concepts from one system are applied to another system? (e.g. variable assignment)

Do systems you currently use provide tutorials and examples? Do you allow students to learn from these?

Should the RaspberryPi be included in the learning process?

Are there restrictions on the software you can install? Is software run from single server or installed on every machine? Do any students “tinker” while programming? How do they do that? If so, is there any difference in gender? Do students experience any anxiety before, during or after carrying out programming tasks? Have students been frustrated with programs that don’t run as expected? In your opinion, to what extent does mathematic aptitude influence a student’s ability to understand programming concepts? Is it required? What educational techniques do you use while teaching computational concepts and programming? Are you aware of the campaign for Computational Thinking to be taught as a more generic transferrable skill, independent of traditional programming or computer science? What are your thoughts on it? Do you have any standard current methods of evaluation other than exams? How useful do you think these are?
Section 2: Future Practices
What do you think would improve the current tools available used teach programming? Are you thinking about using any new tools in the future? What do you think would improve the current teaching methods? Are you thinking about using any new methods of teaching in the future?
Section 3: Demonstration
Could you see any benefit/problems with a system that used paper, visual blocks (like scratch), and text programming (like python). If so, what? You already use websites as a goal to achieve; do you think this is a motivating area for children? If so, why? Would you be interested in evaluating such a system? Do you have any feedback on methods for final evaluation? Which aspects of the demonstration do you think would be most valuable to students’ education? What additions would be required to use this system in a real classroom? If there are any, could you prioritise them? Do you think the tool should contain learning documentation and tutorials, or would you prefer to teach the concepts? Why is this? Do you think any of the following features would be particularly useful/not useful:

- Collaboration features
- Automatic evaluation
- Helper agents
- Tutorials
- Ability to continue at home

Is there anything I should have asked you?

Appendix B

Note: For brevity, only the pre-test is provided, as the post-test and Lesson 2 post-test are identical besides order of questions and variable names.

Pilot Study Questionnaire

Introduction

Welcome to DrawBridge! As part of this session, we will ask you to create some animations. **This is not a test.** Our aim is to find out how well DrawBridge works, and to find out if you think any improvements can be made.

The answers you give here and to the researcher will be kept private. No one will ever know what you say unless you tell them. Your name will never be used.

Before you begin, please answer the questions below. If you answer "yes" to any question, please give more details.

Gender
☐ Boy ☐ Girl
How old are you?
______years old
Year Group
Year _____
Have you made Animations on the Computer before? If so, how?
Have you ever programmed a computer before? If so, how?

If you have programmed, how long have you been programming for?	
Have you ever heard of/used JavaScript?	
Do you enjoy Computing? What are the best and worst parts?	
Which of these languages have you used?	
☐ Python ☐ JavaScript ☐ Java ☐ C++ ☐ Visual Basic ☐ Scratch ☐ Alice ☐ LISP ☐ C# ☐ C	Others:

Pilot Study Pre-Test

Programming Questionnaire (Pre)

Introduction

The following questions are multiple-choice. The results will **not** be used to assess you, and will not be shared with either your teacher, or anyone other than the researcher.

Questions

A1. Which of the following are correct?	
var x = 2;	○
var x == 2;	○
varx = 2;	○
var x <= 2;	○

A2. Which of the following are correct?	
X equals Y	○
X <= y	○
x = y;	○
X == y	○

Q3. Which of the following are correct?	

Play function(y, z){
}
	○

Function play(y, z){
}
	○

Var x = function(y, z){
}
	○

Var x = function(100, 100){
}
	○

Q4. Which of the following run without error?

y = x + 1 + 2;	○
y = x = 1;	○
y = x / 0;	○
Y / x - 1	○

Q5. Which of the following run without error?	

Function x(a, b) {};
X();
	○

X();
Function x(a, b) {};
	○

Var x = function x(a, b) {}
X();
	○

X();
Var x = function x(a, b) {}
	○

Q6. Which of the following run without error?	
;	○
Var x = y;	○
Var x = -1;	○
Var x = 0.01;	○

Pre-Questionnaire

Welcome to DrawBridge! As part of this session, we will ask you to create some animations.

This is not a test – relax ☺

The answers you give here and to the researcher will be kept private.

Name: ______________________________

School: ______________________________

Date Today: ______ / ______ / _____________

Gender: ☐ Boy ☐ Girl

How old are you? ___________ Years old.

Year Group: Year _______

Have you made Animations on the Computer Before? If so, how?

☐ Yes How?

☐ No

☐ Don't Know

Have you ever made your own webpage before? If so, how?

☐ Yes How?

☐ No

☐ Don't Know

Have you ever programmed a computer before? If so, how?

☐ Yes How?

☐ No

☐ Don't Know

If you have programmed, how long have you been programming for?

☐ Less than a year

☐ 1 year

☐ 2-3 years

☐ More than 3 years

What is your skill level at using computers?

☐ Poor Strengths/Weaknesses?

☐ Fair

☐ Good

☐ Very Good

☐ Excellent

Do you enjoy using computers? What are the best and worst parts?

☐	Hate it	Best and Worst Parts:
☐	Not great	
☐	Don't mind	
☐	It's ok	
☐	Love it	

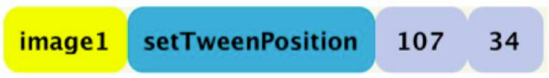
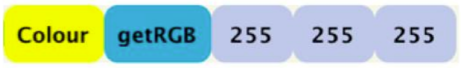
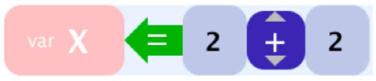
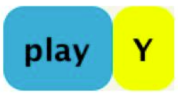
Pre/Post-Test

1. Do these pieces of text look correct?

www.google.com	Yes ☐	No ☐	Not Sure ☐
user@gmail.com	Yes ☐	No ☐	Not Sure ☐
variable x = 2;	Yes ☐	No ☐	Not Sure ☐
var x == 2;	Yes ☐	No ☐	Not Sure ☐
varx = 2;	Yes ☐	No ☐	Not Sure ☐
var x = 2	Yes ☐	No ☐	Not Sure ☐
var x <= 2;	Yes ☐	No ☐	Not Sure ☐
var x = 2;	Yes ☐	No ☐	Not Sure ☐

setImagePosition 100;	Yes ☐	No ☐	Not Sure ☐
setImagePosition(100);	Yes ☐	No ☐	Not Sure ☐

2. **Can you fill in the missing pieces?**

Blocks	Code
 image1 setTweenPosition 107 34	
	`setTimeToDestination(500);`
 Colour getRGB 255 255 255	
	`loadImage(2, 541, 173, 238, 168);`
 var X = 2 + 2	
	`document.getElementById("myCanvas");`
 play Y	

	`var X = 2 / 2;`

Tasks – Group TF

DrawBridge Tasks

Today you are going to be using a version of DrawBridge, which just focuses on programming code.

Task 1:

Using the panel on the left, try to type the text below into DrawBridge.

```
var x = 2;
var y = 3;

function swap(a, b){
    var c = a;
    a = b;
    b = c;
}

swap(x, y);
```

Notice what happens on the right screen. Are there blocks that appear the same as the code you have written?

Task 2:

1. Can you see that the blocks on the right are the same as the text on the left?
2. Now delete the code you wrote in Task 1.
3. Now try to make the block version of the code by dragging blocks into the right hand screen from the bottom.
4. Now check the text produced on the left against the text given in Task 1. Is it the same?
5. Write the differences in the box below, along with the reasons why the text might have been different.

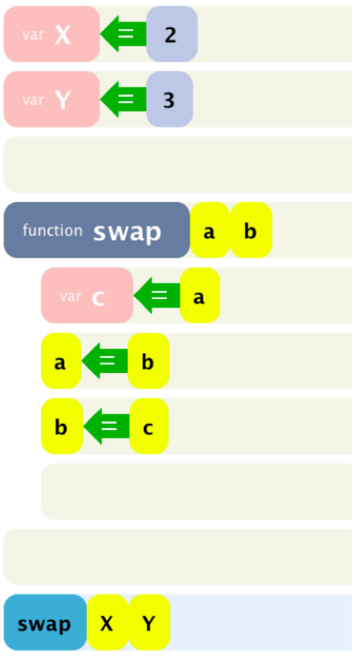
Tasks – Group VF

DrawBridge Tasks

Today you are going to be using a version of DrawBridge, which just focuses on programming code.

Task 1:

Using the panel on the left, type the following code into DrawBridge:



```
var X = 2
var Y = 3

function swap a b
    var c = a
    a = b
    b = c


swap X Y
```

Notice what happens on the right screen. Is the text that appears the same as the blocks you have created?

Task 2:

6. Try to see how the text on the right is the same as the blocks on the left.
7. Remove the blocks you added in Task 1 (by dragging the back to the bottom).
8. Now try to make the text version of the code that was generated by the blocks in Task 1.
9. Now check the blocks produced on the left against the blocks given in Task 1. Are they the same?
10. Write the differences in the box below, along with the reasons why the text might have been different.

Tasks –

Group C-TF

DrawBridge Tasks

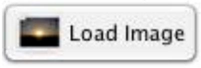
If you follow the steps in each task, you can create an animation that you can view in a web browser and share with your friends! If you have any trouble during the tasks, please find the researcher, or your teacher, and ask them. Try not to ask your friends.

Task 1

You will need: Coloured pencils, paper and DrawBridge.

Step 1. Make sure you have downloaded the correct copy of DrawBridge to your computer. The researcher should have helped you to do this already.

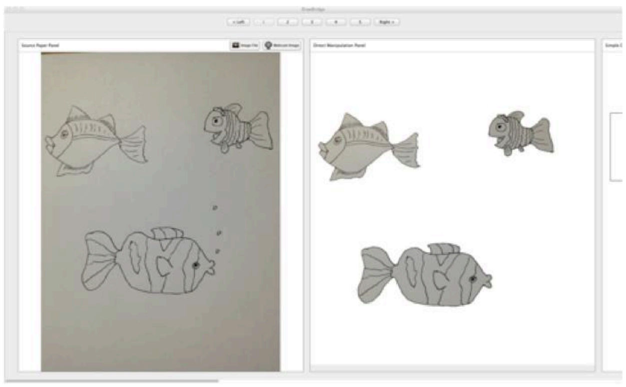
Step 2. Click the "Load Image" button in DrawBridge.



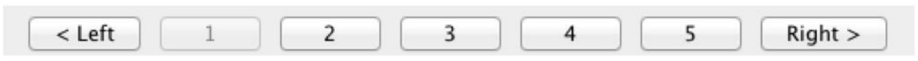
Step 3. Find the image that has been loaded onto your computer by the researcher in the file chooser.

Step 4. Double click on the image

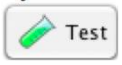
Step 5. Your characters should appear separately on the right hand side of the screen (similar to below, but with your characters).



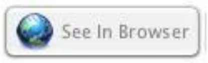
Step 6. Click the right arrow (in the centre of the screen, near the top) to move through the DrawBridge screens. Try not to change anything just yet.



Step 7. Click the "test" button to see preview your characters.



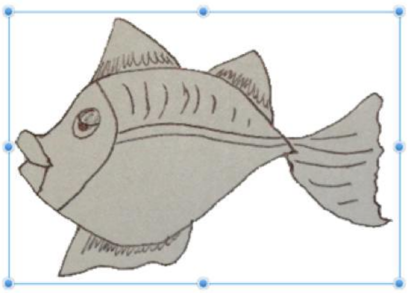
Step 8. Click the “See in Browser” button to see your characters in a web browser!



Task 2

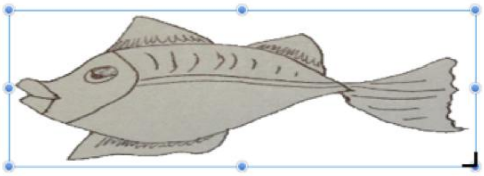
Step 1. Using the navigation buttons, click “1” to go to first two screens in DrawBridge.

Step 2. In the second screen, try clicking on one of your characters. It should become selected (like below).

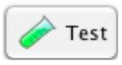


Step 3. Click and drag the character to a new position.

Step 4. Use the resizing handles to change the size of your character

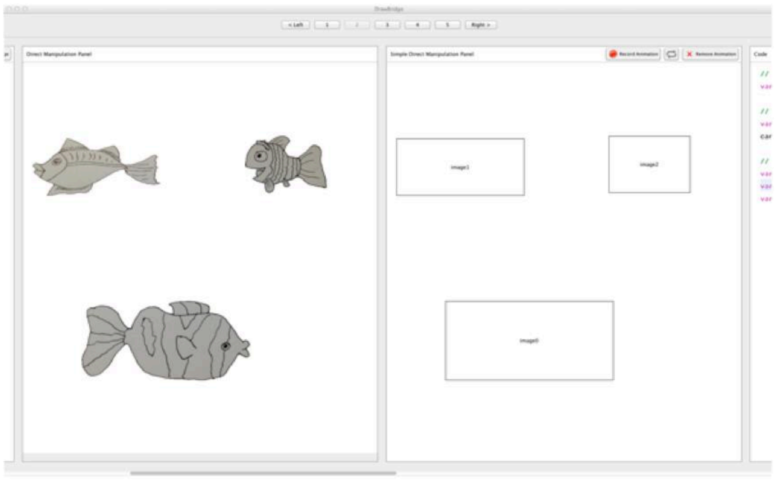


Step 5. Using the navigation buttons, move to the final DrawBridge screen, and click the “Test” button.

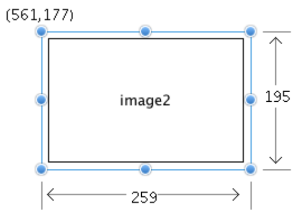


Task 3

Step 1. Using the navigation buttons, move to the "2" to see the second and third screens in DrawBridge. They should look like this (with your characters):

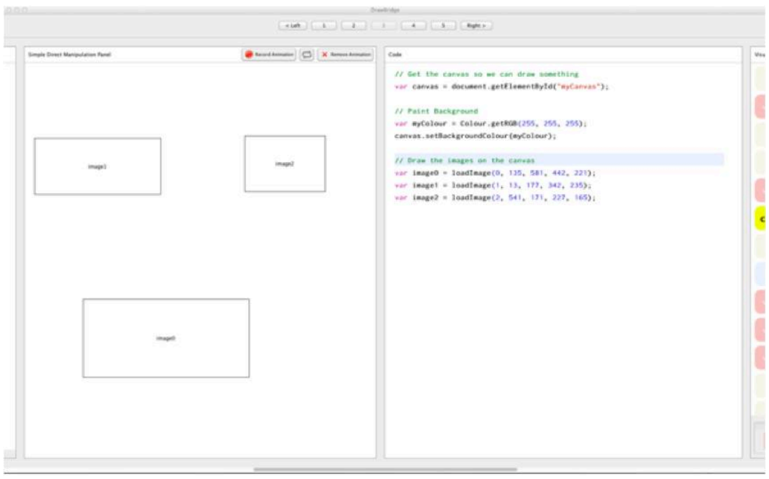


Step 2. The rectangles on the right are how the computer sees your characters. Click on one to see its properties.



Step 3. Try moving and resizing a rectangle. Notice that your character on the left moves and resizes at the same time.

Step 4. Using the navigation buttons, move right one place (to show the third and fourth screens). You should see coloured blocks on the right hand side (like in the image below).



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 135, 581, 442, 221);
var image1 = loadImage(1, 13, 177, 342, 235);
var image2 = loadImage(2, 541, 171, 227, 165);
```

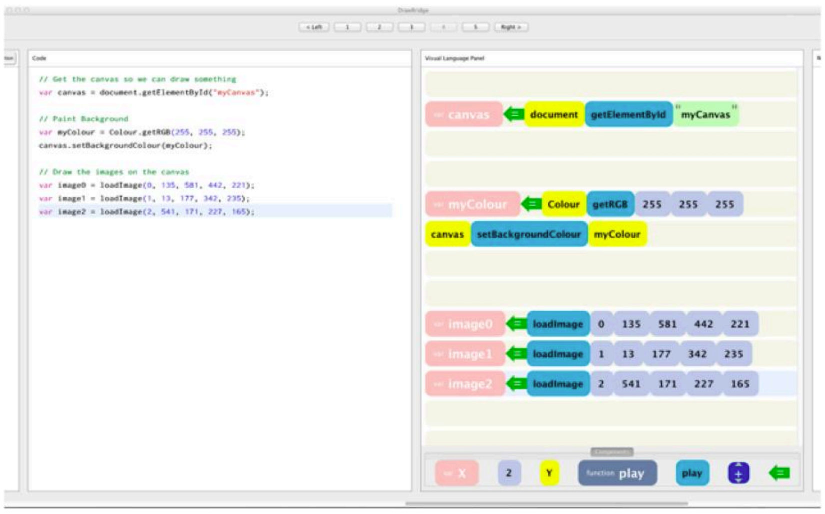
Step 5. Next, create an animation by clicking the “Record” button. Move and resize one of your characters, then click the “stop” button.

Step 6. Notice how the text changes when you move the rectangles when recording.

Step 7. Using the arrows, move to the last screen. Click the “Test” button.

Task 4

Step 1. Using the navigation buttons, move to the fourth and fifth screens in DrawBridge. They should look like this (with your characters):



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 135, 581, 442, 221);
var image1 = loadImage(1, 13, 177, 342, 235);
var image2 = loadImage(2, 541, 171, 227, 165);
```

Step 2. You should see some "programming code" blocks on the right that make your characters animate. Try slowing down your animation by increasing the number (time in milliseconds) it animates for.

```
image1.setTweenPosition(94, 336);
image1.setTweenSize(347, 241);
setTimeToDestination(673);
```

Step 3. Using the arrows, go to the last screen in DrawBridge and click the "Test" button. Your animation should be slower.

Step 4. Using the arrows in DrawBridge, go back to the fourth and fifth screens. Try adding your own text to make an extra step in the animation.

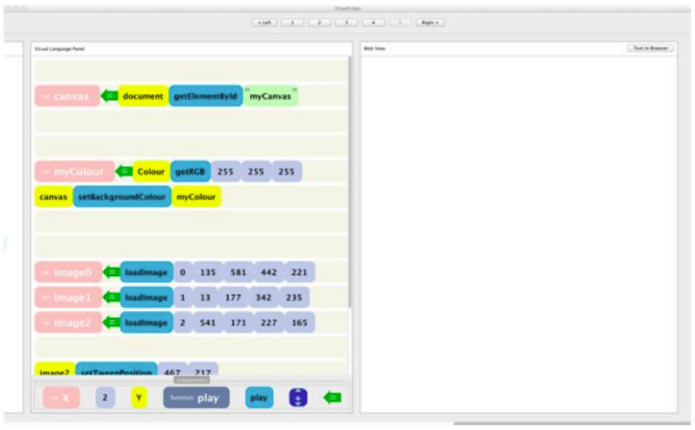
```
image1.setTweenPosition(94, 336);
image1.setTweenSize(347, 241);
setTimeToDestination(673);
```

Step 5. When you make changes, or add text, you should see the blocks on the right hand screen change to match what you have done.

Step 6. Using the arrows, go to the last screen in DrawBridge and click the "Test" button. You should see the extra step in your animation.

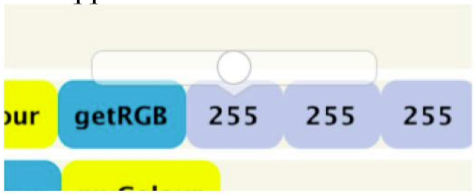
Task 5

Step 1. Using the arrows, move to the last screens in DrawBridge. They should look like this (with your characters):



```
canvas      document getElementById "myCanvas"
myColour    Colour getRGB 255 255 255
canvas setBackgroundColour myColour
image0      loadImage 0 135 581 442 221
image1      loadImage 1 13 177 342 235
image2      loadImage 2 541 171 227 165
image2 setTweenPosition 467 217
```

Step 2. On the right hand screen, hover your mouse over a number you would like to change. A slider will appear like this:

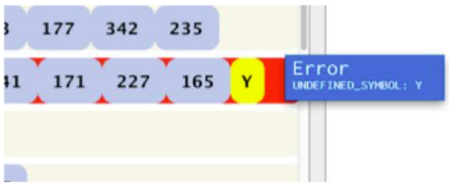


Step 3. Move the slider left and right. The number will change. The animation preview on the right will also change to show your changes.

Step 4. Click the “Test” button to preview the animation.

Step 5. Back in DrawBridge, on the left hand screen, try editing and adding new blocks.

Step 6. When your code is wrong, you should see a red dot at the side of the text. You can hover over the red dot to see what the problem is.



Error
UNDEFINED_SYMBOL: Y

Tasks – Group C-VF

DrawBridge Tasks

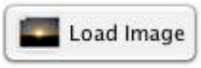
If you follow the steps in each task, you can create an animation that you can view in a web browser and share with your friends! If you have any trouble during the tasks, please find the researcher, or your teacher, and ask them. Try not to ask your friends.

Task 1

You will need: Coloured pencils, paper and DrawBridge.

Step 9. Make sure you have downloaded the correct copy of DrawBridge to your computer. The researcher should have helped you to do this already.

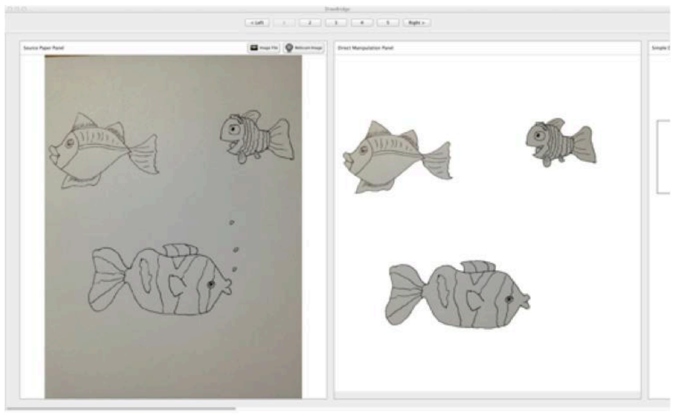
Step 10. Click the "Load Image" button in DrawBridge



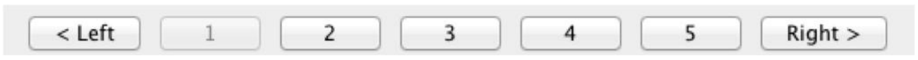
Step 11. Find the image that has been loaded onto your computer by the researcher.

Step 12. Double click on the image

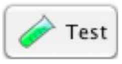
Step 13. Your characters should appear separately on the right hand side of the screen (similar to below, but with your characters).



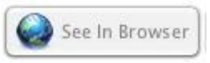
Step 14. Click the right navigation button (in the center of the screen, near the top) to move through the DrawBridge screens. Try not to change anything just yet.



Step 15. Click the "Test" button to see a preview of your animation.



Step 16. Click the “See in Browser” button to see your characters in a web browser!

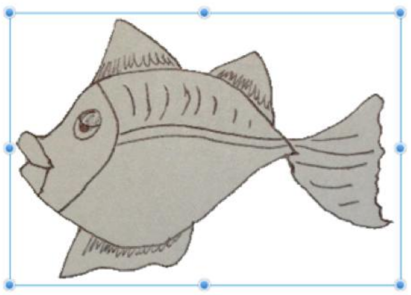


Task 2

You will need: DrawBridge.

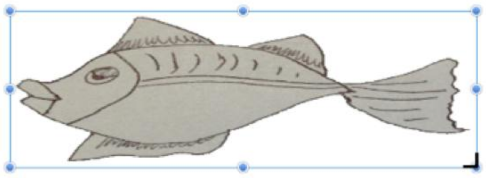
Step 6. Using the navigation buttons, move back to the first two screens in DrawBridge.

Step 7. In the second screen, try clicking on one of your characters. It should become selected (like below).

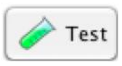


Step 8. Click and drag the character to a new position.

Step 9. Use the resizing handles to change the size of your character



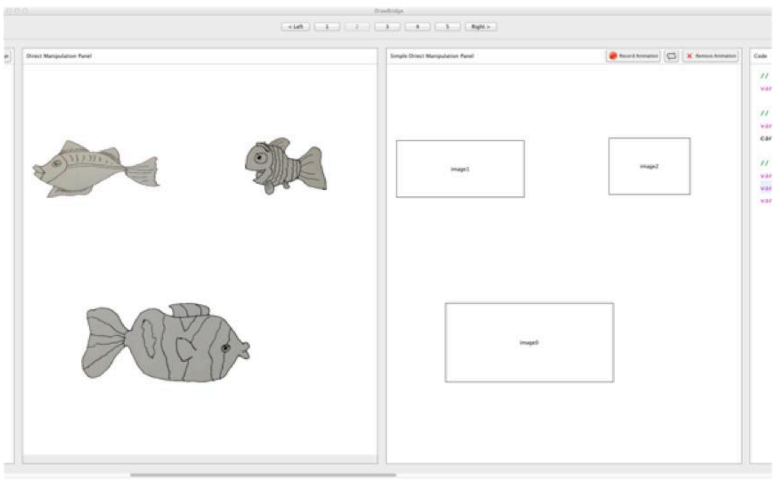
Step 10. Using the navigation buttons, move to the final DrawBridge screen, and click the “Test” button.



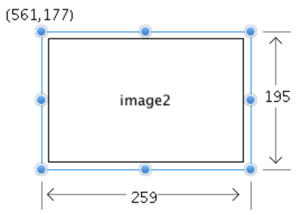
Task 3

You will need: DrawBridge.

Step 8. Using the navigation buttons, move to the second and third screens in DrawBridge. They should look like this (with your characters):

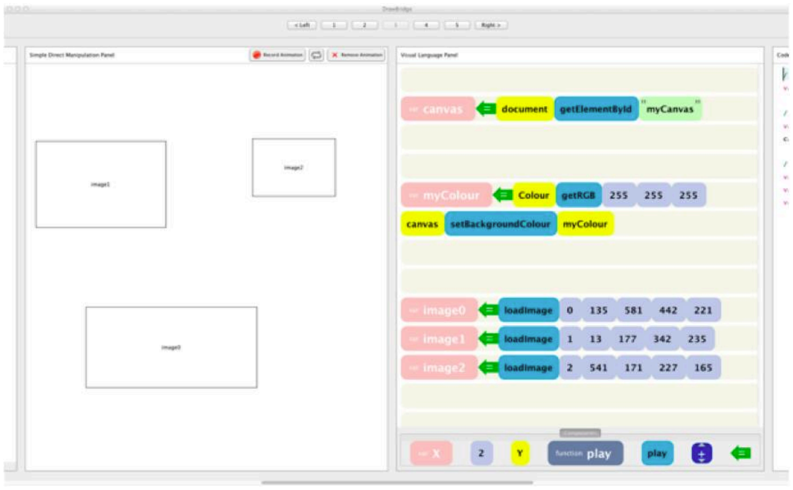


Step 9. The rectangles on the right are how the computer sees your characters. Click on one to see its properties.



Step 10. Try moving and resizing a rectangle. Notice that your character on the left moves and resizes at the same time.

Step 11. Using the navigation buttons, move right one place (to show the third and fourth screens). You should see coloured blocks on the right hand side (like in the image below).



```
canvas = document getElementById "myCanvas"

myColour = Colour getRGB 255 255 255
canvas setBackgroundColour myColour

image0 = loadImage 0 135 581 442 221
image1 = loadImage 1 13 177 342 235
image2 = loadImage 2 541 171 227 165
```

Step 12. Next, create an animation by clicking the “Record” button. Move and resize one of your characters, then click the “stop” button.

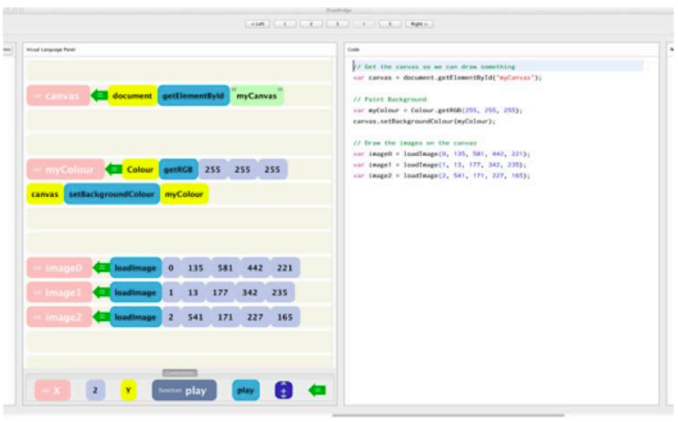
Step 13. Notice how the visual blocks change when you move the rectangles when recording.

Step 14. Using the navigation buttons, move to the last screen. Click the “Test” button to preview your animation.

Task 4

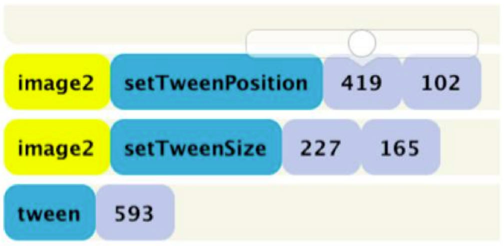
You will need: DrawBridge.

Step 7. Using the navigation buttons, move to the fourth and fifth screens in DrawBridge. They should look like this (with your characters):



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 135, 581, 442, 221);
var image1 = loadImage(1, 13, 177, 342, 235);
var image2 = loadImage(2, 541, 171, 227, 165);
```

Step 8. You should see some “programming code” blocks on the left that make your characters animate. Try slowing down your animation by increasing the number (amount of time it “tweens” for). You can do this by editing the text, or using the “dialler”.



Step 9. Using the navigation buttons, go to the last screen in DrawBridge and click the “Test” button. Your animation should be slower.

Test

Step 10. Using the navigation buttons in DrawBridge, go back to the fourth and fifth screens. Try adding your own blocks (by dragging them from the palette at the bottom) to make an extra step in the animation.

Step 11. When you make changes, or add blocks, you should see the text on the right hand screen change to match what you have done.

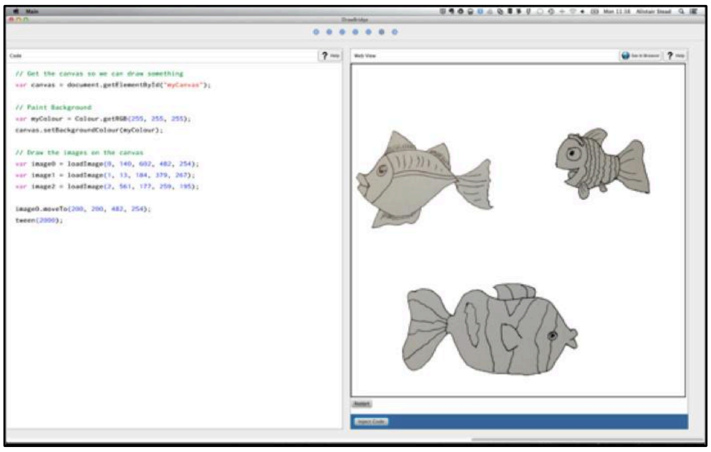
Step 12. Using the navigation buttons, go to the last screen in DrawBridge and click the “Test” button. You should see the extra step in your animation.

Test

Task 5

You will need: DrawBridge.

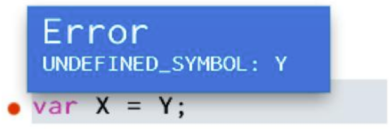
Step 7. Using the navigation buttons, move to the last screens in DrawBridge. They should look like this (with your characters):



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadImage(0, 140, 602, 482, 254);
var image1 = loadImage(1, 13, 184, 379, 267);
var image2 = loadImage(2, 561, 177, 259, 195);

image0.moveTo(200, 200, 482, 254);
tween(2000);
```

Step 8. On the left hand screen, try editing and adding new lines of code.

Step 9. When your code is wrong, you should see a red dot at the side of the text. You can hover over the red dot to see what the problem is.



```
Error
UNDEFINED_SYMBOL: Y
var X = Y;
```

Post-Questionnaire

1. **Overall, how much have you enjoyed using DrawBridge?**

☹	🙁	🙂	😊	😃
Hated it	Disliked it	Didn't mind it	Liked it	Loved it
☐	☐	☐	☐	☐

2. **Would you like to use DrawBridge again?**

☹	🙁	🙂	😊	😃
Never	Not really	Don't mind	OK	Definitely
☐	☐	☐	☐	☐

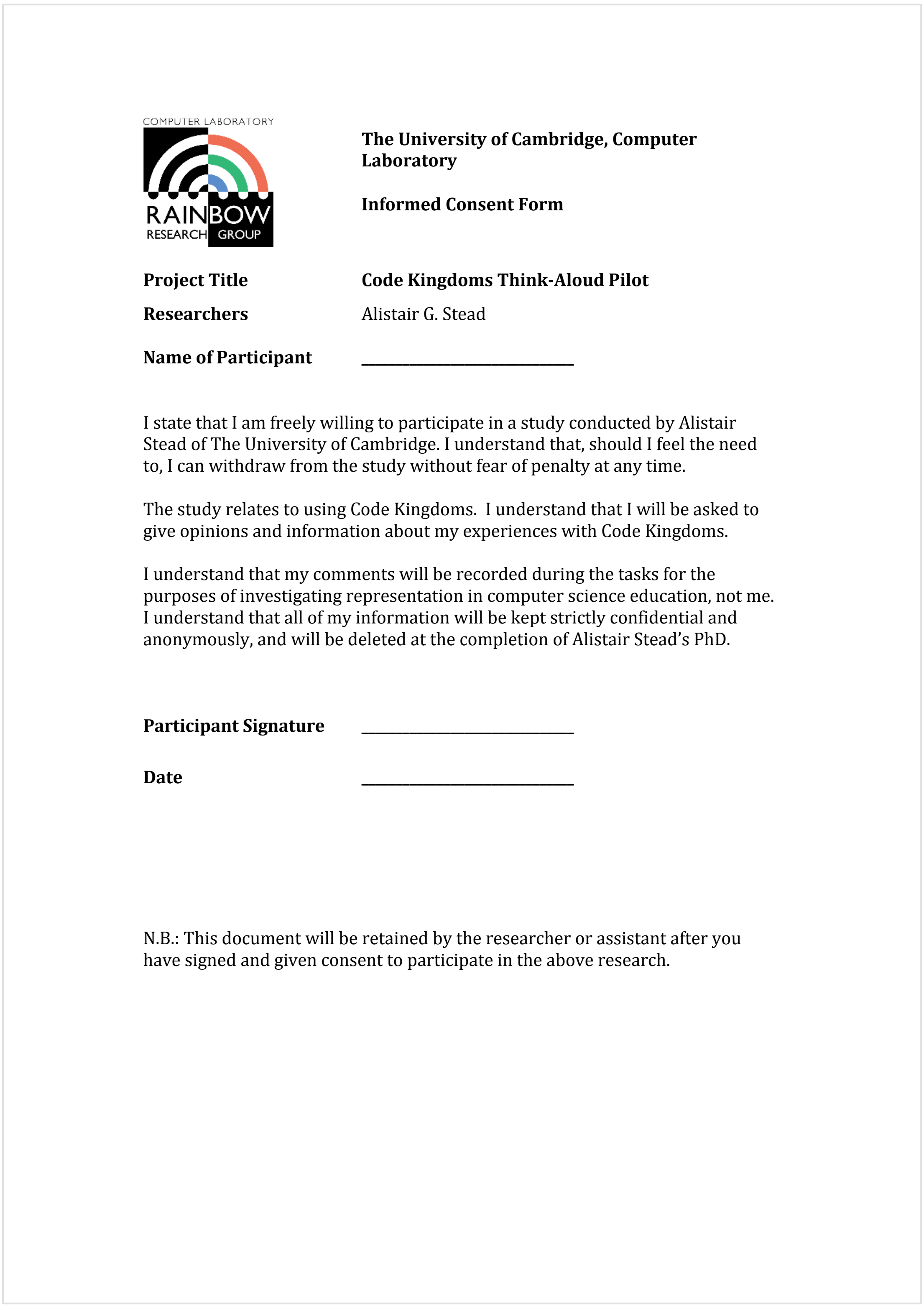
3. **What are the three best things about DrawBridge?**

4. **What are the three worst things about DrawBridge?**

5. **How can we improve DrawBridge?**

Appendix C

Consent Form



COMPUTER LABORATORY
RAINBOW RESEARCH GROUP

The University of Cambridge, Computer Laboratory

Informed Consent Form

Project Title	**Code Kingdoms Think-Aloud Pilot**
Researchers	Alistair G. Stead
Name of Participant	____________________________

I state that I am freely willing to participate in a study conducted by Alistair Stead of The University of Cambridge. I understand that, should I feel the need to, I can withdraw from the study without fear of penalty at any time.

The study relates to using Code Kingdoms. I understand that I will be asked to give opinions and information about my experiences with Code Kingdoms.

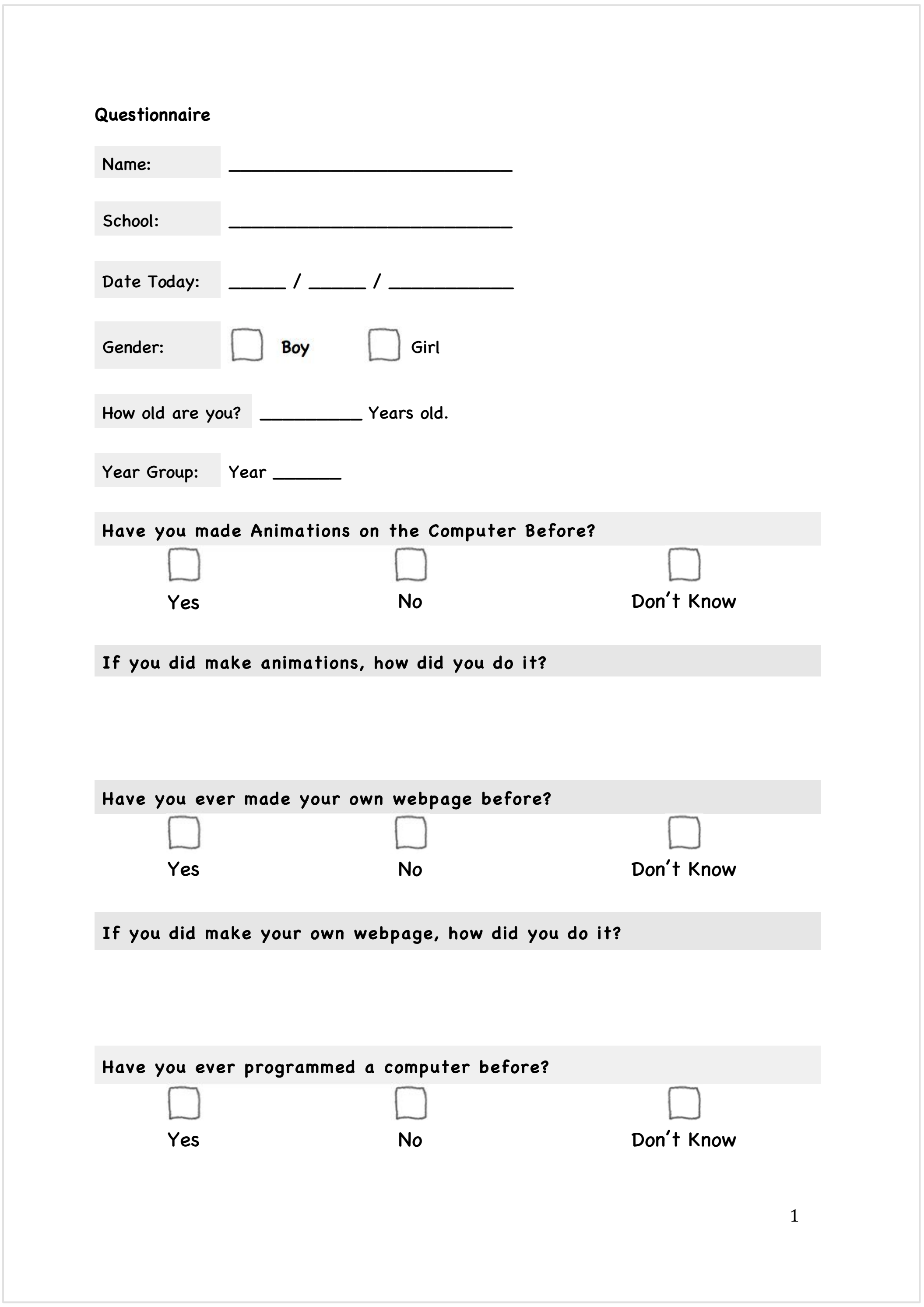
I understand that my comments will be recorded during the tasks for the purposes of investigating representation in computer science education, not me. I understand that all of my information will be kept strictly confidential and anonymously, and will be deleted at the completion of Alistair Stead's PhD.

Participant Signature ____________________________

Date ____________________________

N.B.: This document will be retained by the researcher or assistant after you have signed and given consent to participate in the above research.

Background Questionnaire



Questionnaire

Name: ______________________________

School: ______________________________

Date Today: ______ / ______ / ____________

Gender: ☐ Boy ☐ Girl

How old are you? __________ Years old.

Year Group: Year _______

Have you made Animations on the Computer Before?

☐ Yes ☐ No ☐ Don't Know

If you did make animations, how did you do it?

Have you ever made your own webpage before?

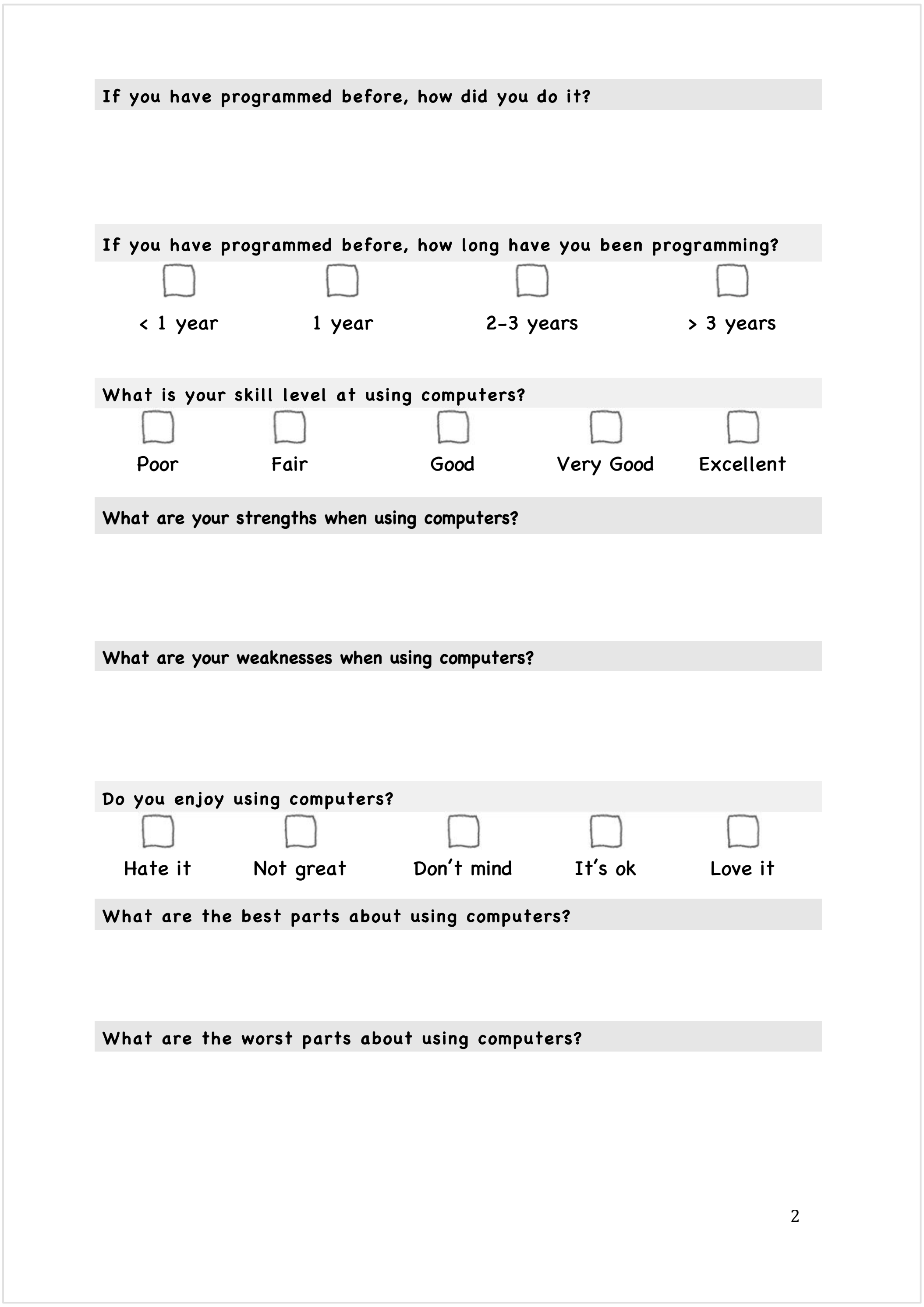
☐ Yes ☐ No ☐ Don't Know

If you did make your own webpage, how did you do it?

Have you ever programmed a computer before?

☐ Yes ☐ No ☐ Don't Know

1



If you have programmed before, how did you do it?

If you have programmed before, how long have you been programming?

☐	☐	☐	☐
< 1 year	1 year	2-3 years	> 3 years

What is your skill level at using computers?

☐	☐	☐	☐	☐
Poor	Fair	Good	Very Good	Excellent

What are your strengths when using computers?

What are your weaknesses when using computers?

Do you enjoy using computers?

☐	☐	☐	☐	☐
Hate it	Not great	Don't mind	It's ok	Love it

What are the best parts about using computers?

What are the worst parts about using computers?

2

Assessment: Code Writing

Code Kingdoms
WHAT HAVE YOU LEARNT?

Write the statement that does the following things in Code Kingdoms

Tick off your progress as you go

Example

q) Make something called "myVariable"

```
var myVariable;
```

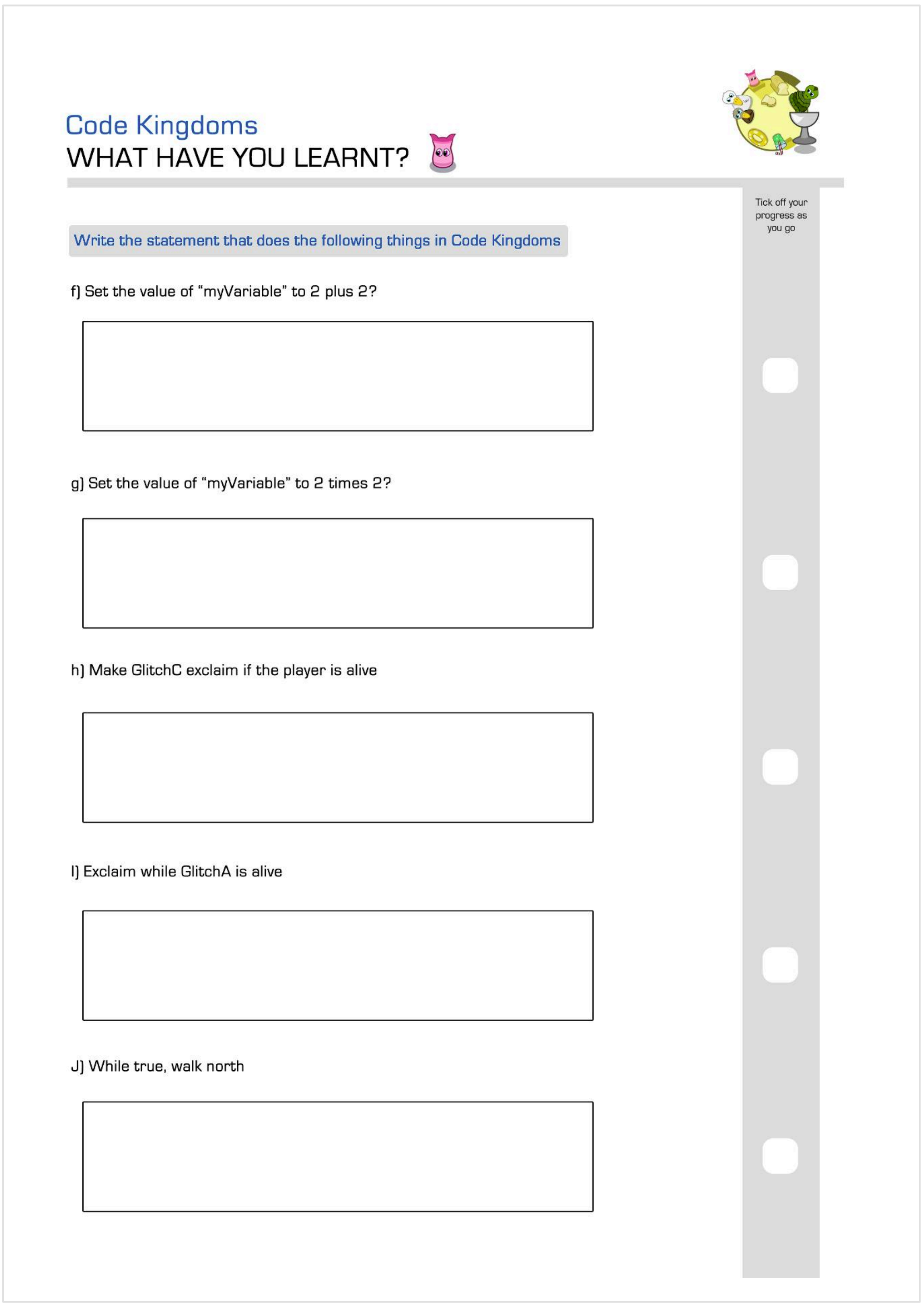
✓

a) Give "myVariable" the value 2

b) Change the "age" of "GlitchB" to 10

c) Make helperA say "hello!"

d) Stop GlitchC



Code Kingdoms

WHAT HAVE YOU LEARNT?

Tick off your progress as you go

Write the statement that does the following things in Code Kingdoms

f) Set the value of "myVariable" to 2 plus 2?

g) Set the value of "myVariable" to 2 times 2?

h) Make GlitchC exclaim if the player is alive

I) Exclaim while GlitchA is alive

J) While true, walk north

Assessment: MCQ Questions

Code Kingdoms

WHAT HAVE YOU LEARNT?

Tick the correct answer for each question:

Tick off your progress as you go

Example

q) Make something called "myVariable"

- ☐ var myVariable
- ☐ myVariable var
- ☑ var myVariable;
- ☐ ;myVariable var

✓

a) Give "myVariable" the value 2

- ☐ myVariable = 2
- ☐ myVariable == 2
- ☐ myVariable = 2;
- ☐ myVariable == 2;

b) Change the "age" of "GlitchB" to 10

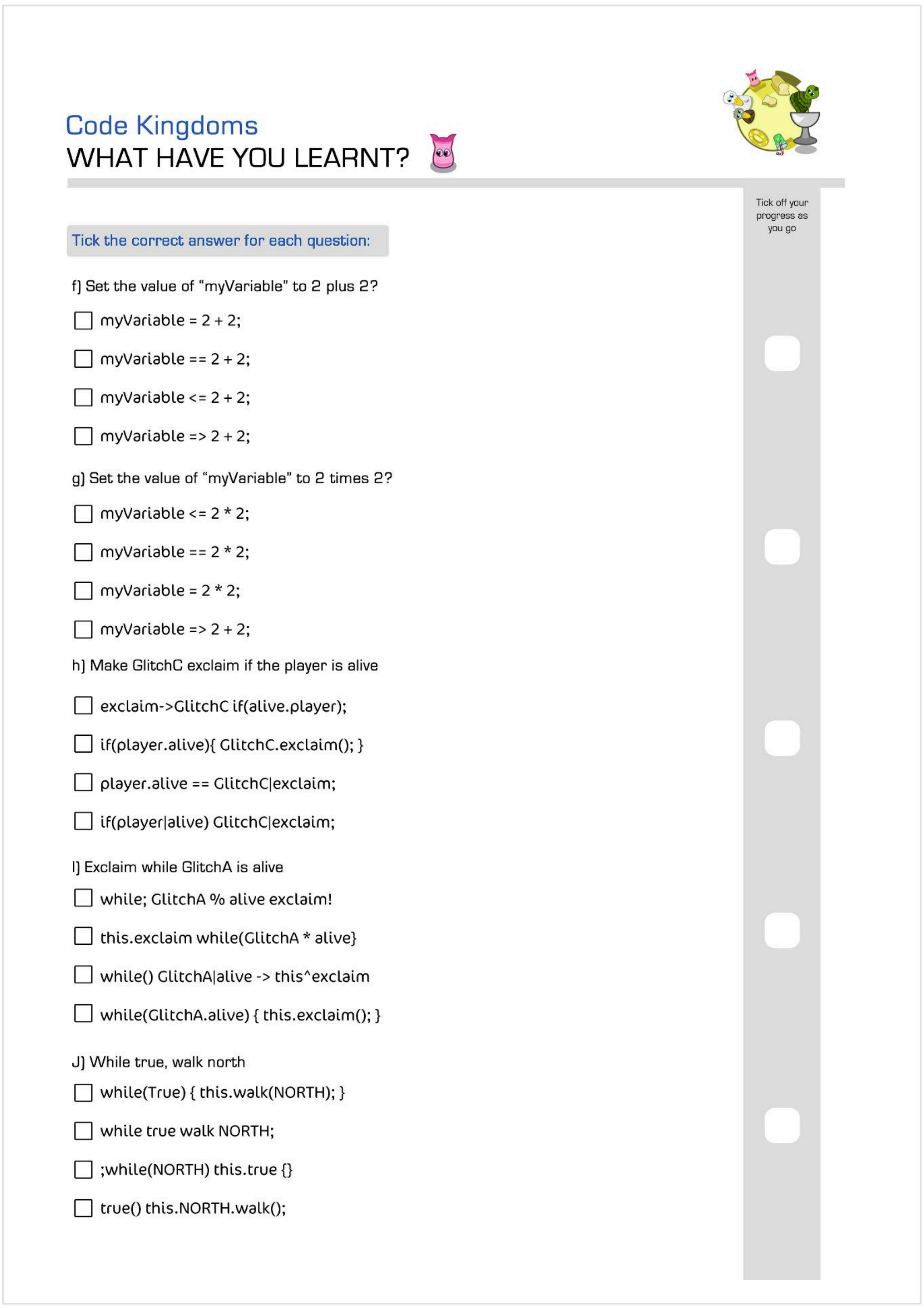
- ☐ GlitchB.age = 10;
- ☐ age.GlitchB = 10
- ☐ GlitchB->age = 10;
- ☐ GlitchB.age <= 10

c) Make helperA say "hello!"

- ☐ helperA.say(hello!)
- ☐ helperA|hello!
- ☐ helperA.say("hello!");
- ☐ say("hello!")->helperA

d) Stop GlitchC

- ☐ Stop->GlitchC();
- ☐ ;stop(GlitchC;)
- ☐ GlitchC.stop();
- ☐ GlitchC<-stop;



Code Kingdoms

WHAT HAVE YOU LEARNT?

Tick off your progress as you go

Tick the correct answer for each question:

f] Set the value of "myVariable" to 2 plus 2?

☐ myVariable = 2 + 2;

☐ myVariable == 2 + 2;

☐ myVariable <= 2 + 2;

☐ myVariable => 2 + 2;

g] Set the value of "myVariable" to 2 times 2?

☐ myVariable <= 2 * 2;

☐ myVariable == 2 * 2;

☐ myVariable = 2 * 2;

☐ myVariable => 2 + 2;

h] Make GlitchC exclaim if the player is alive

☐ exclaim->GlitchC if(alive.player);

☐ if(player.alive){ GlitchC.exclaim(); }

☐ player.alive == GlitchC|exclaim;

☐ if(player|alive) GlitchC|exclaim;

I] Exclaim while GlitchA is alive

☐ while; GlitchA % alive exclaim!

☐ this.exclaim while(GlitchA * alive}

☐ while() GlitchA|alive -> this^exclaim

☐ while(GlitchA.alive) { this.exclaim(); }

J] While true, walk north

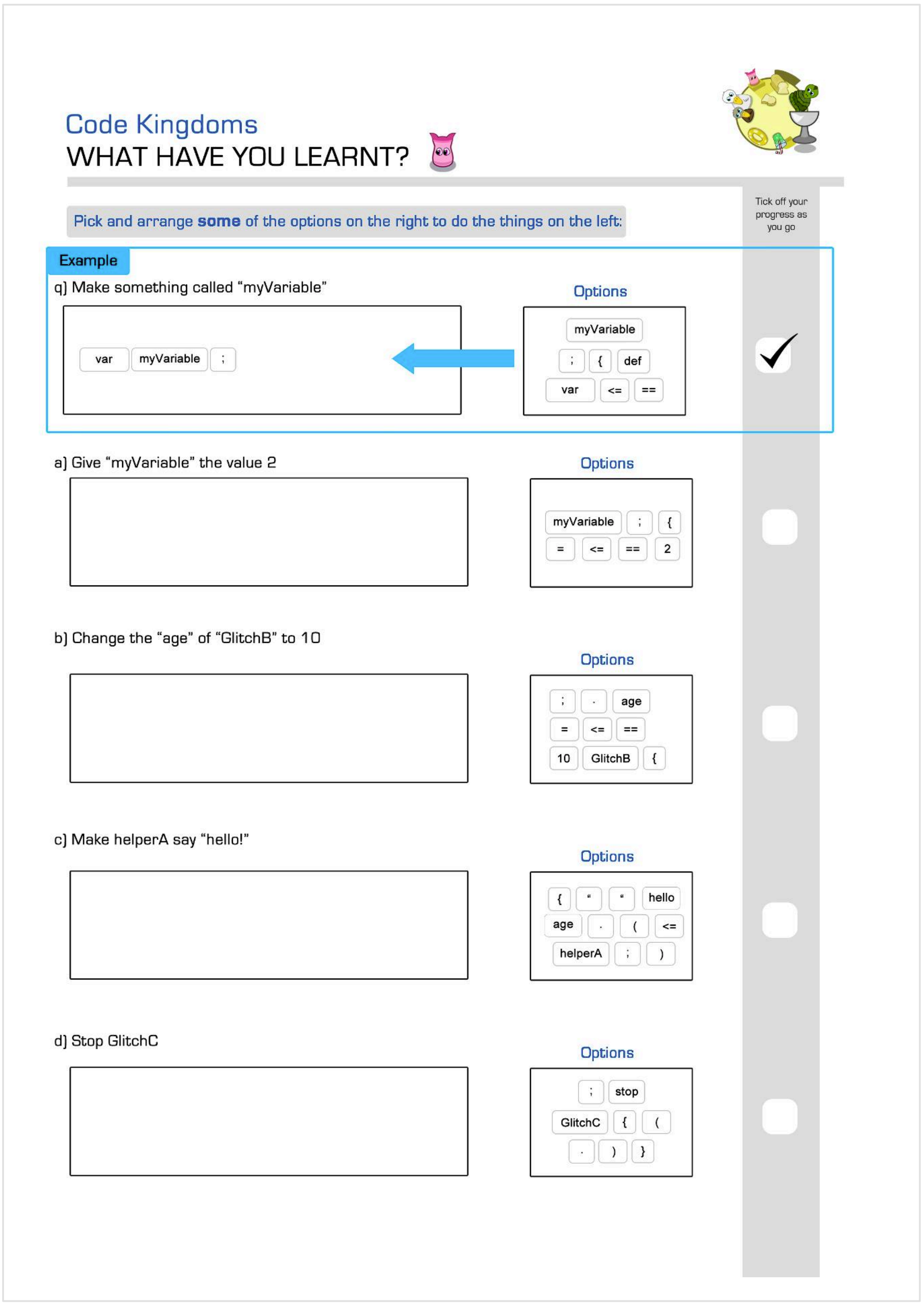
☐ while(True) { this.walk(NORTH); }

☐ while true walk NORTH;

☐ ;while(NORTH) this.true {}

☐ true() this.NORTH.walk();

Assessments: Adapted Parsons Problems



Code Kingdoms

WHAT HAVE YOU LEARNT?

Pick and arrange **some** of the options on the right to do the things on the left:

Tick off your progress as you go

Example

q) Make something called "myVariable"

```
var myVariable ;
```

Options

```
myVariable
; { def
var <= ==
```

✓

a) Give "myVariable" the value 2

Options

```
myVariable ; {
= <= == 2
```

b) Change the "age" of "GlitchB" to 10

Options

```
; . age
= <= ==
10 GlitchB {
```

c) Make helperA say "hello!"

Options

```
{ " " hello
age . ( <=
helperA ; )
```

d) Stop GlitchC

Options

```
; stop
GlitchC { (
. ) }
```

Code Kingdoms
WHAT HAVE YOU LEARNT?

Tick off your progress as you go

f) Set the value of "myVariable" to 2 plus 2?

Options

```
; myVariable 2
+ = 2
. { }
```

g) Set the value of "myVariable" to 2 times 2?

Options

```
* ; myVariable
2 = 2
) (
```

h) Make GlitchC exclaim if the player is alive

Options

```
GlitchC player (
if ) ( .
exclaim alive )
. { } ; ==
```

I) Exclaim while GlitchA is alive

Options

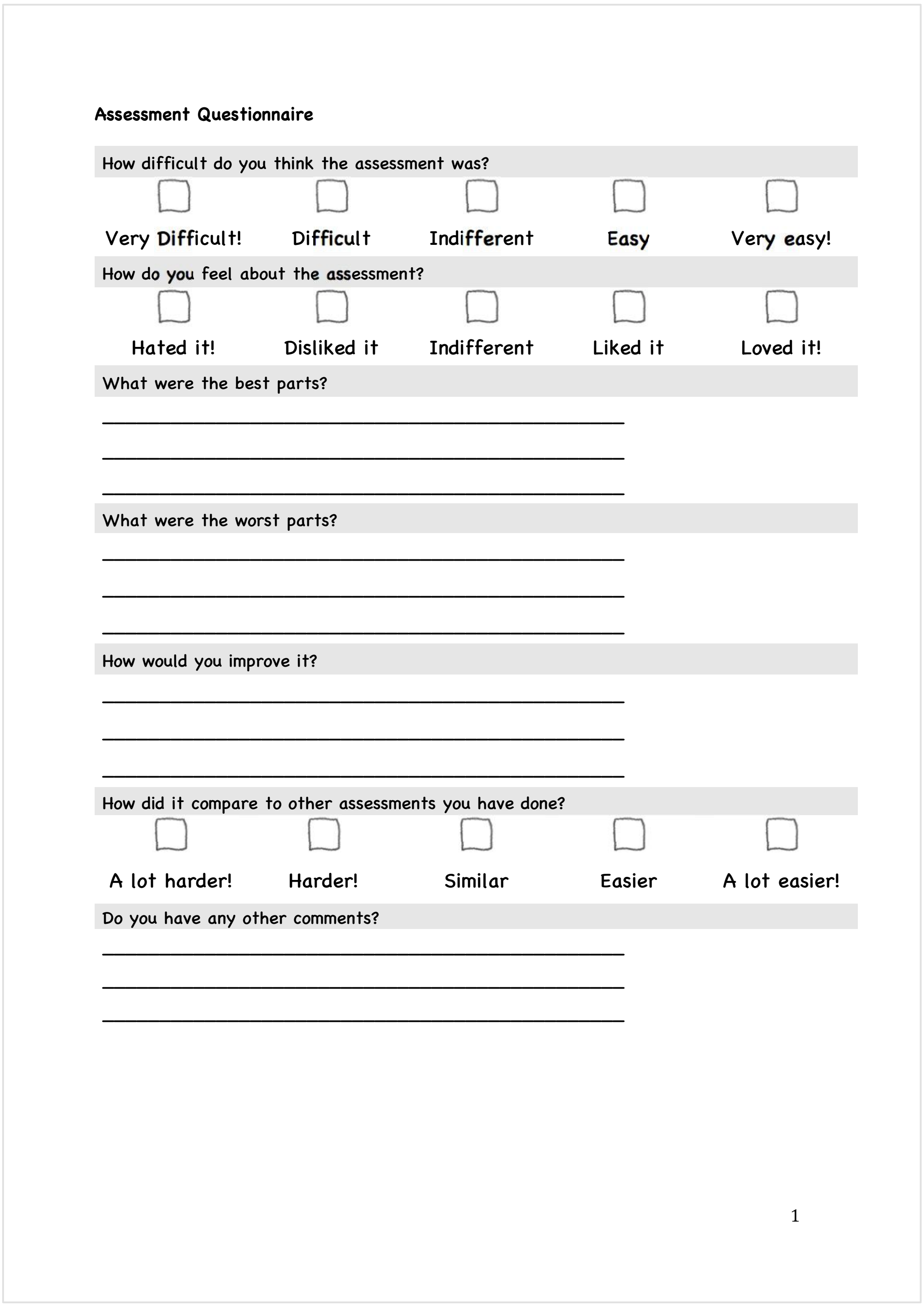
```
== while { . )
= exclaim } (
alive GlitchA while .
this ) ( ; ||
```

J) While true, walk north

Options

```
== while { . )
= this } (
walk True NORTH
this ) ( ;
```

Post-Assessment Questionnaire



Assessment Questionnaire

How difficult do you think the assessment was?

☐	☐	☐	☐	☐
Very Difficult!	Difficult	Indifferent	Easy	Very easy!

How do you feel about the assessment?

☐	☐	☐	☐	☐
Hated it!	Disliked it	Indifferent	Liked it	Loved it!

What were the best parts?

____

____

____

What were the worst parts?

____

____

____

How would you improve it?

____

____

____

How did it compare to other assessments you have done?

☐	☐	☐	☐	☐
A lot harder!	Harder!	Similar	Easier	A lot easier!

Do you have any other comments?

____

____

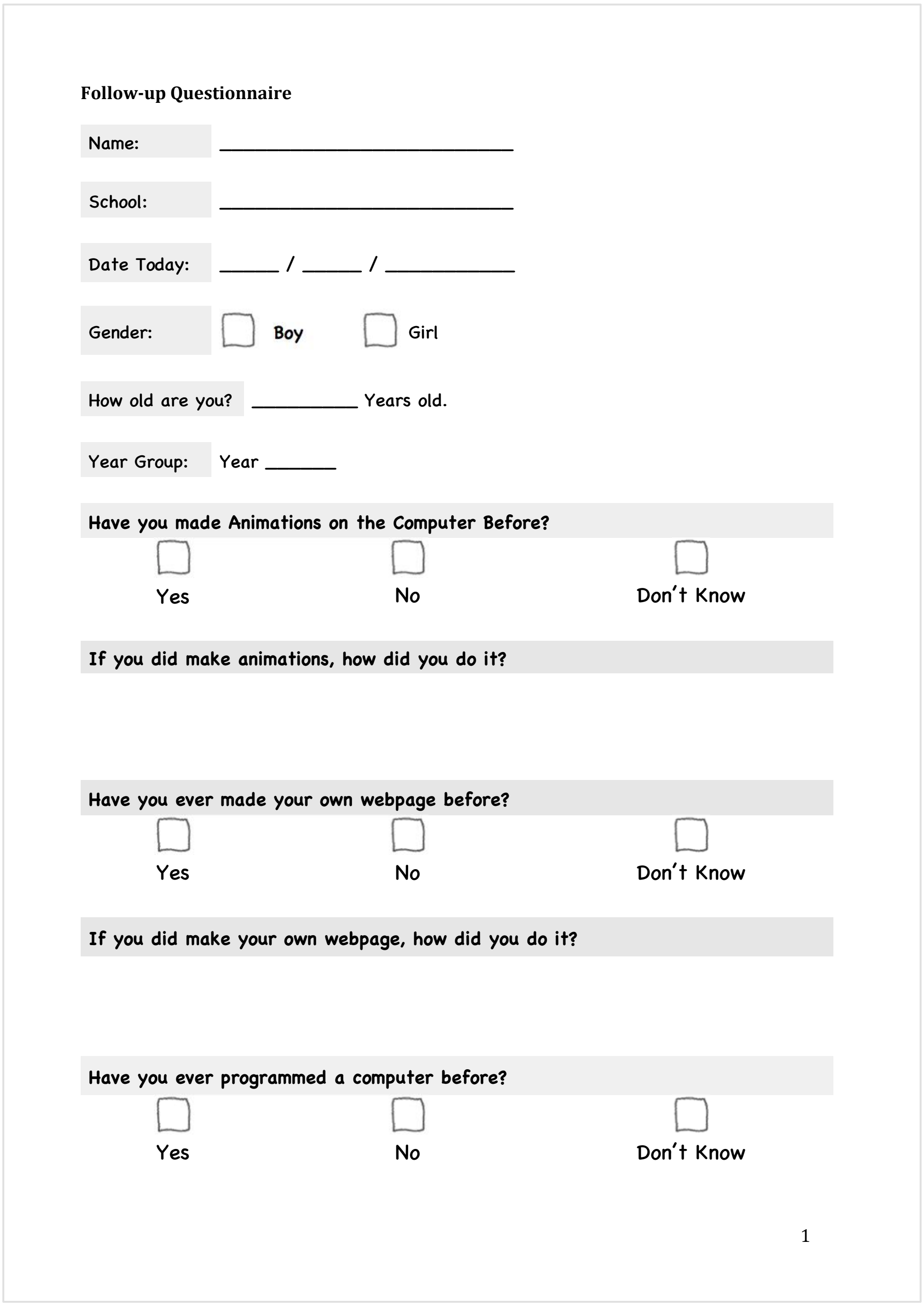
____

1

Appendix D

Note: For brevity, only the pre-study questionnaire is provided, as the pilot study questionnaire was identical.

Pre-Study Questionnaire



Follow-up Questionnaire

Name: ______________________________

School: ______________________________

Date Today: ______ / ______ / ____________

Gender: ☐ Boy ☐ Girl

How old are you? ___________ Years old.

Year Group: Year _______

Have you made Animations on the Computer Before?

☐ Yes ☐ No ☐ Don't Know

If you did make animations, how did you do it?

Have you ever made your own webpage before?

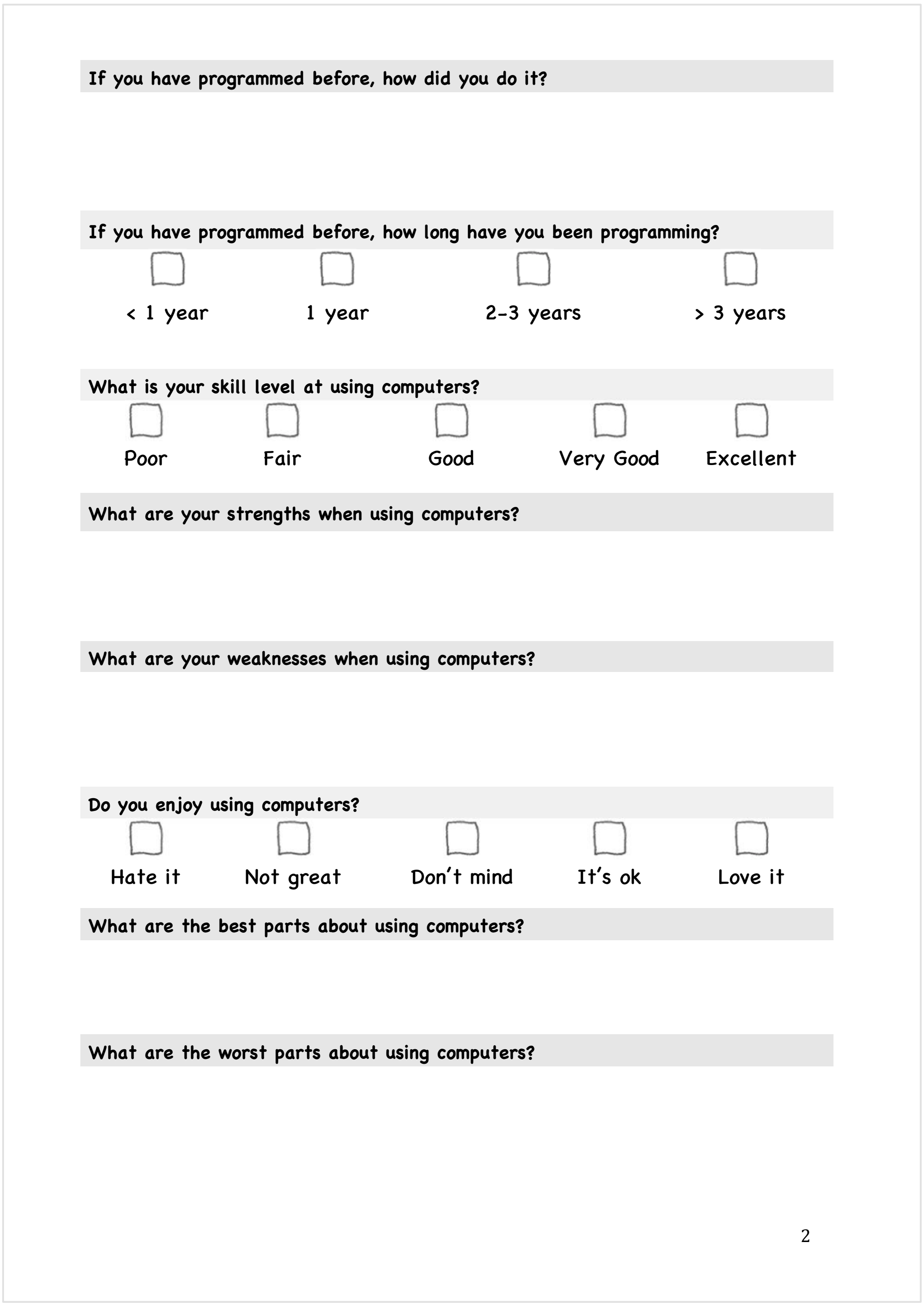
☐ Yes ☐ No ☐ Don't Know

If you did make your own webpage, how did you do it?

Have you ever programmed a computer before?

☐ Yes ☐ No ☐ Don't Know

1



If you have programmed before, how did you do it?

If you have programmed before, how long have you been programming?

☐	☐	☐	☐
< 1 year	1 year	2-3 years	> 3 years

What is your skill level at using computers?

☐	☐	☐	☐	☐
Poor	Fair	Good	Very Good	Excellent

What are your strengths when using computers?

What are your weaknesses when using computers?

Do you enjoy using computers?

☐	☐	☐	☐	☐
Hate it	Not great	Don't mind	It's ok	Love it

What are the best parts about using computers?

What are the worst parts about using computers?

2

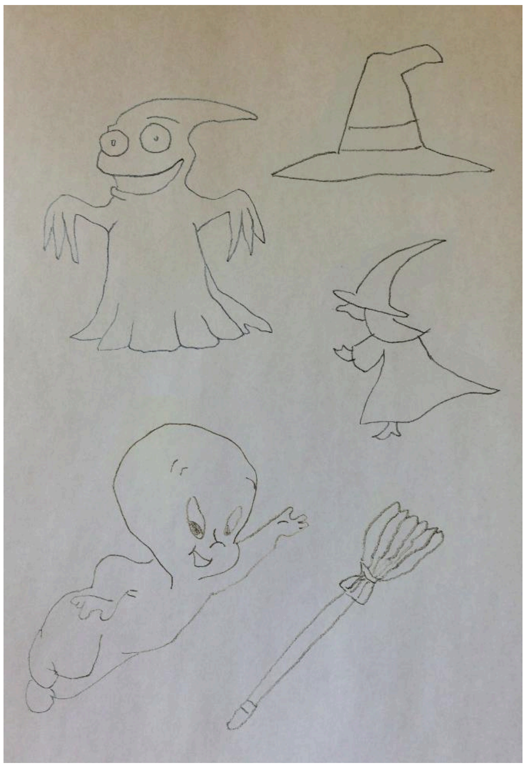
Worksheets Lesson 1

DrawBridge

Halloween Horrors

Introduction:

Today you are going to make an animation for Halloween using scary images that you draw on paper. To do this, you will be using Drawbridge, an piece of experimental software built at the University of Cambridge.



Required:
- DrawBridge Drawing Paper
- Pencil/Pen
- DrawBridge
- Sense of humour

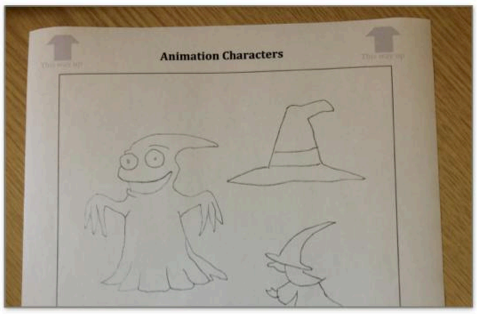
1

Halloween Horrors

Tick off your progress as you go

Step1: Drawing Halloween Characters on Paper

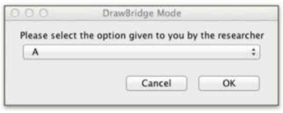
1. On your DrawBridge drawing paper, draw some scary characters you would like to animate in today's session. Make sure you draw with the paper in portrait orientation.



2. Take a picture of your characters and load them onto your computer (the researcher will help you with this)

3. Double click on DrawBridge.jar - the researcher will tell you where to find this.

4. Use the dialog box to select which group you are in. You can find out your group by looking for the group sheet given to you at the start of the lesson, or by asking the researcher.



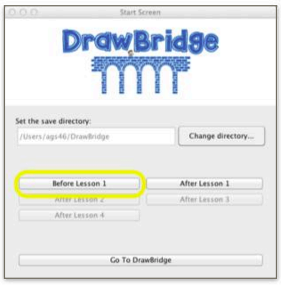
DrawBridge Mode

Please select the option given to you by the researcher

A

5. Use the DrawBridge start screen to open the "Before Lesson 1" screen, which will check to see how much of DrawBridge you already know.

6. Complete the parsons questions and move to the next step.



Start Screen

Set the save directory:

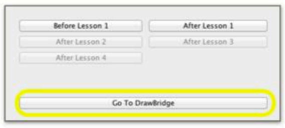
/Users/ags46/DrawBridge

2

Step 2: Adding a Scary Background

Tick off your progress as you go

1. On the DrawBridge Start Screen, Click "Go To DrawBridge".

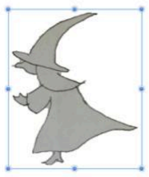


2. Welcome to DrawBridge! Use the Image File button to load the scary image you drew!

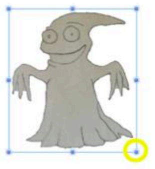
3. When your image loads on the left, you should see your individual characters on the right. Use the Change Background button to select the scary background.

Things to Watch Out For:
DrawBridge has left and right buttons near the top that you can use to move between panels. The number buttons will take you to specific positions in DrawBridge.

4. Move your characters by clicking and dragging with your mouse on the "Character Panel"



5. Resize your characters using the handles on the selected character. Try making one really big character, one medium sized character, and one small character.



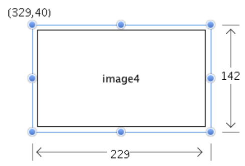
3

Halloween Horrors

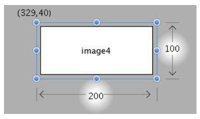
Step 3: Making scary characters specific sizes

Tick off your progress as you go

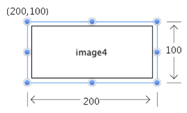
1. Click the Right > navigation button twice. The"Animation Panel" should be on the left, and the "Block Panel" should be on the right. Select one of the boxes that you can see there. It should look like this:



2. The boxes represent your characters. If you move/resize them, your characters will move/resize at the same time. Try resizing one of the boxes so that it has a width of 200 and a height of 100.



3. Try moving one of the boxes to x 200 and y 100.



Things to Watch Out For:
As you move or resize the boxes, look at the "Block Panel". The blocks change to match what you are doing! Try to find the blocks that change when you move your boxes!

Halloween Horrors

Tick off your progress as you go

Step 4: Coding using Blocks!

We're going to use the blocks to change our characters! Blocks are small round rectangles that can be dragged into a position to make the computer do things. There are different types of blocks. You can tell the type by looking at the colour.

var image1

1. Find out which block relate to your biggest character and look at the "setLocation" function. In the example below, the blocks set the location of image0 to and x (or horizontal) position of 226 and a y (or vertical) position of 506

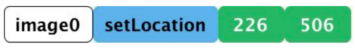


image0 setLocation 226 506

2. Hover over the green x position block (226 in the example above). You should see a slider, which you can move left and right to change the value. Change the value to 300 using the slider.

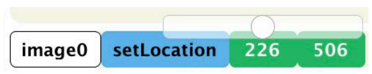


image0 setLocation 226 506

3. Change the height of the same image to 300 by changing the second green block next in the "setSize" function.

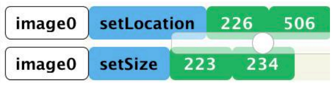


image0 setLocation 226 506
image0 setSize 223 234

4. Instead of using the slider, change the width of the image by double clicking on the block, and typing 150.

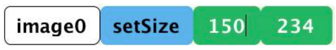


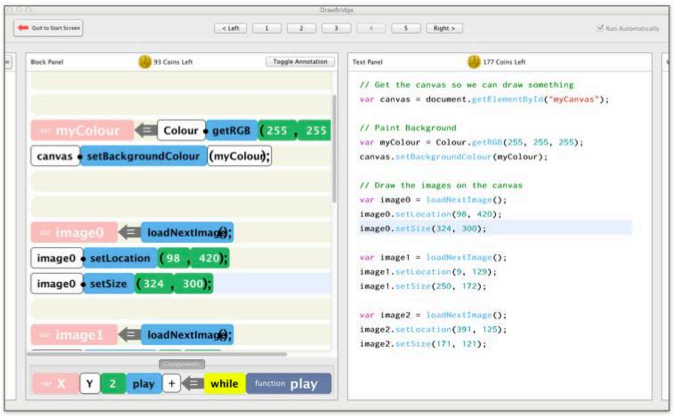
image0 setSize 150 234

Things to Watch Out For:
You can drag new blocks onto the block editor using the block palette at the bottom of the panel. To remove blocks, drag the block onto the block palette!

Step 6: Coding using Text!

We're going to use some JavaScript to change properties of our characters. We'll use it later to add animations, too!

1. Click the Right > button so that the "Block Panel" is on the left, and the "Text Panel" is on the right. You should be able to see some JavaScript code, which has the same meaning as the blocks you have just been using.



```
// Get the canvas so we can draw something
var canvas = document.getElementById("myCanvas");

// Paint Background
var myColour = Colour.getRGB(255, 255, 255);
canvas.setBackgroundColour(myColour);

// Draw the images on the canvas
var image0 = loadNextImage();
image0.setLocation(98, 420);
image0.setSize(324, 300);

var image1 = loadNextImage();
image1.setLocation(9, 129);
image1.setSize(250, 172);

var image2 = loadNextImage();
image2.setLocation(391, 125);
image2.setSize(171, 121);
```

2. On the left, change a height block to 221 by using the slider. You should see some of the code change when you change the blocks.

image0 setSize 150 221

```
image0.setSize(150, 221);
```

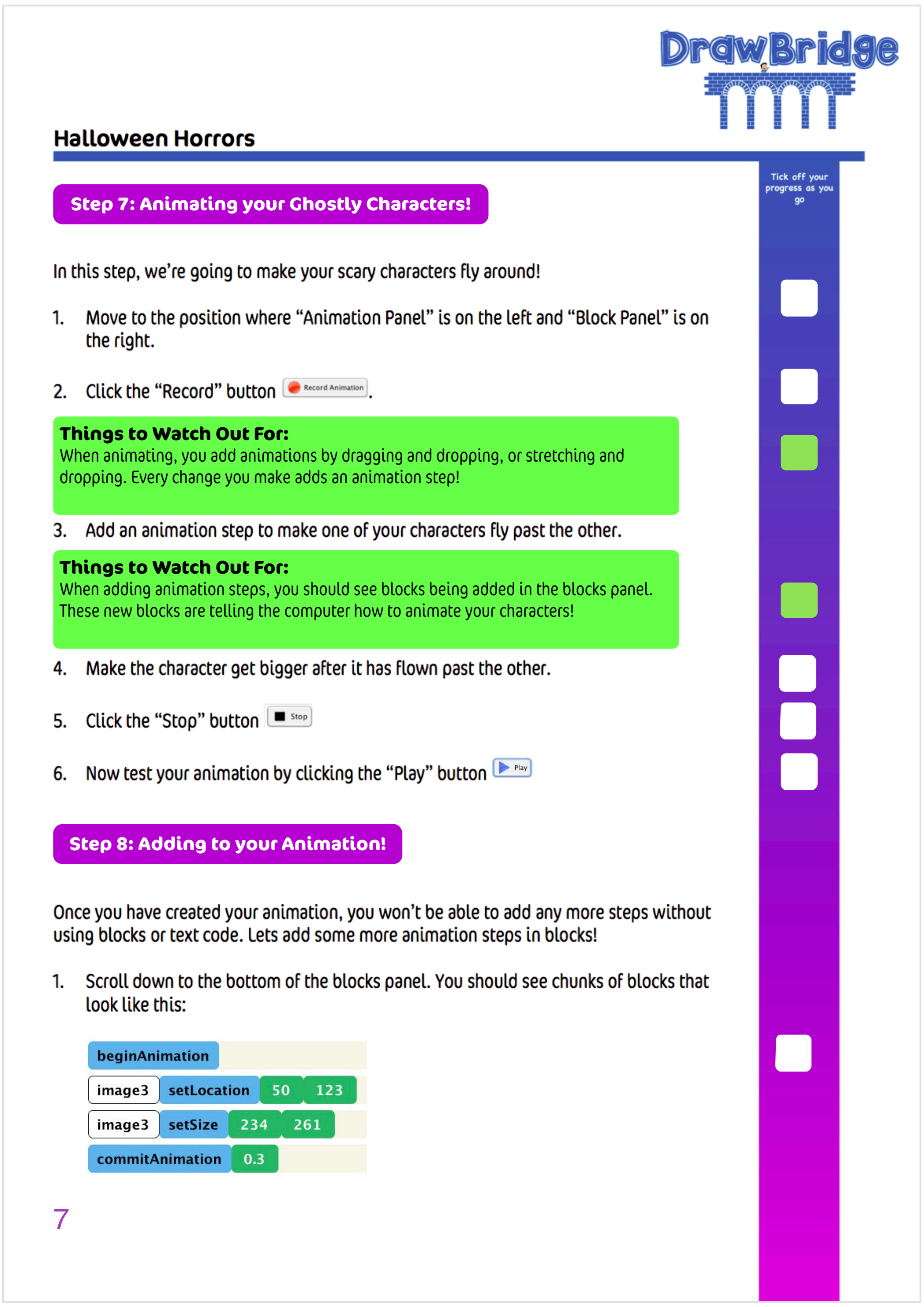
3. Now change the JavaScript (text) height value in the "Text Panel" to 300. Do the blocks change too?

image0 setSize (324 , 300);

```
image0.setSize(324, 300);
```

4. Click the Right > button again, so you see the "Web View" panel on the right. You can test your work by clicking "test" Test

5. Click Test In Browser to view your work on the web. Now go back to DrawBridge and return to the "Animation Panel".



DrawBridge

Halloween Horrors

Tick off your progress as you go

Step 7: Animating your Ghostly Characters!

In this step, we're going to make your scary characters fly around!

1. Move to the position where "Animation Panel" is on the left and "Block Panel" is on the right.

2. Click the "Record" button ● Record Animation.

Things to Watch Out For:
When animating, you add animations by dragging and dropping, or stretching and dropping. Every change you make adds an animation step!

3. Add an animation step to make one of your characters fly past the other.

Things to Watch Out For:
When adding animation steps, you should see blocks being added in the blocks panel. These new blocks are telling the computer how to animate your characters!

4. Make the character get bigger after it has flown past the other.

5. Click the "Stop" button ■ Stop

6. Now test your animation by clicking the "Play" button ▶ Play

Step 8: Adding to your Animation!

Once you have created your animation, you won't be able to add any more steps without using blocks or text code. Lets add some more animation steps in blocks!

1. Scroll down to the bottom of the blocks panel. You should see chunks of blocks that look like this:

```
beginAnimation
image3 setLocation 50 123
image3 setSize 234 261
commitAnimation 0.3
```

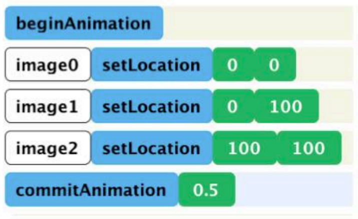
7

Halloween Horrors

Tick off your progress as you go

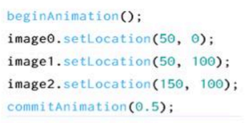
Things to Watch Out For:
Each chunk of blocks is responsible for one animation step. It starts with "beginAnimation" to let the computer know you are doing animation. It finishes with

2. Try making your own new animation step at the bottom of the blocks panel, by dragging blocks onto the panel from the palette, and changing them to match below:



```
beginAnimation
image0 setLocation 0 0
image1 setLocation 0 100
image2 setLocation 100 100
commitAnimation 0.5
```

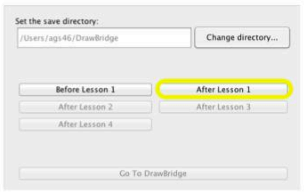
3. Now go to the "Text Panel" and do the same thing in text to make the text below:



```
beginAnimation();
image0.setLocation(50, 0);
image1.setLocation(50, 100);
image2.setLocation(150, 100);
commitAnimation(0.5);
```

4. Experiment by adding as many animation steps as you like and testing them by going to the "WebView" panel and clicking "Test".

5. When you're finished. Click the "Quit to start screen" button to go back to the DrawBridge start screen.



Set the save directory:
/Users/ags46/DrawBridge

6. Complete the questions and then close DrawBridge.

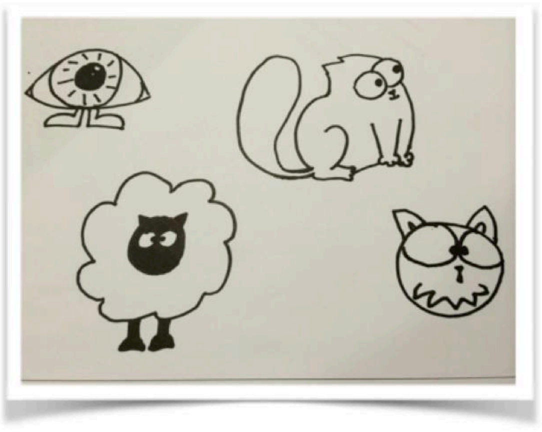
Worksheets Lesson 2

DrawBridge

More About Coding

Introduction:

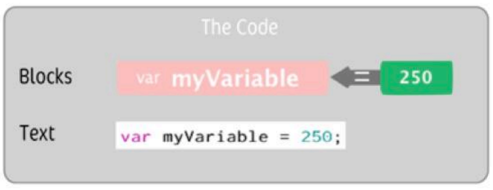
Today we are going to look into the structure of the code in DrawBridge more closely to find out what code is made of, and more importantly, how we can use it to make kick ass animations! This set of reference sheets gives tells you about the code, and why it looks the way it does.



Variables

What is a variable?

A variable is storage for a value (either a number or text). We can give the variable "myVariable" the value 250 below using the code in the box. Think of it like a bucket where you can store things.



```
var myVariable = 250;
```

You can change a variable, or use it as a Input, or parameter for a function.

1

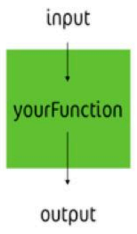
- Try making a variable called "hogwartsHouses" equal to 4.

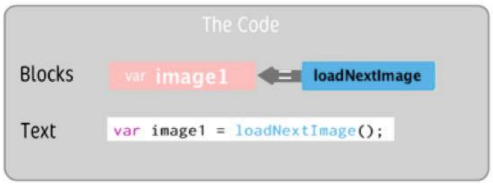
More About Coding

Functions

What is a function?

A function is piece of code that you can reuse. To make functions useful, they can act differently if you give them different inputs. You can think of them as little computers - they take some information, go away and process it, and then give you some information back. There can be no inputs, or lots of inputs - we put them in between brackets.





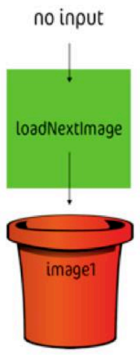
```
var image1 = loadNextImage();
```

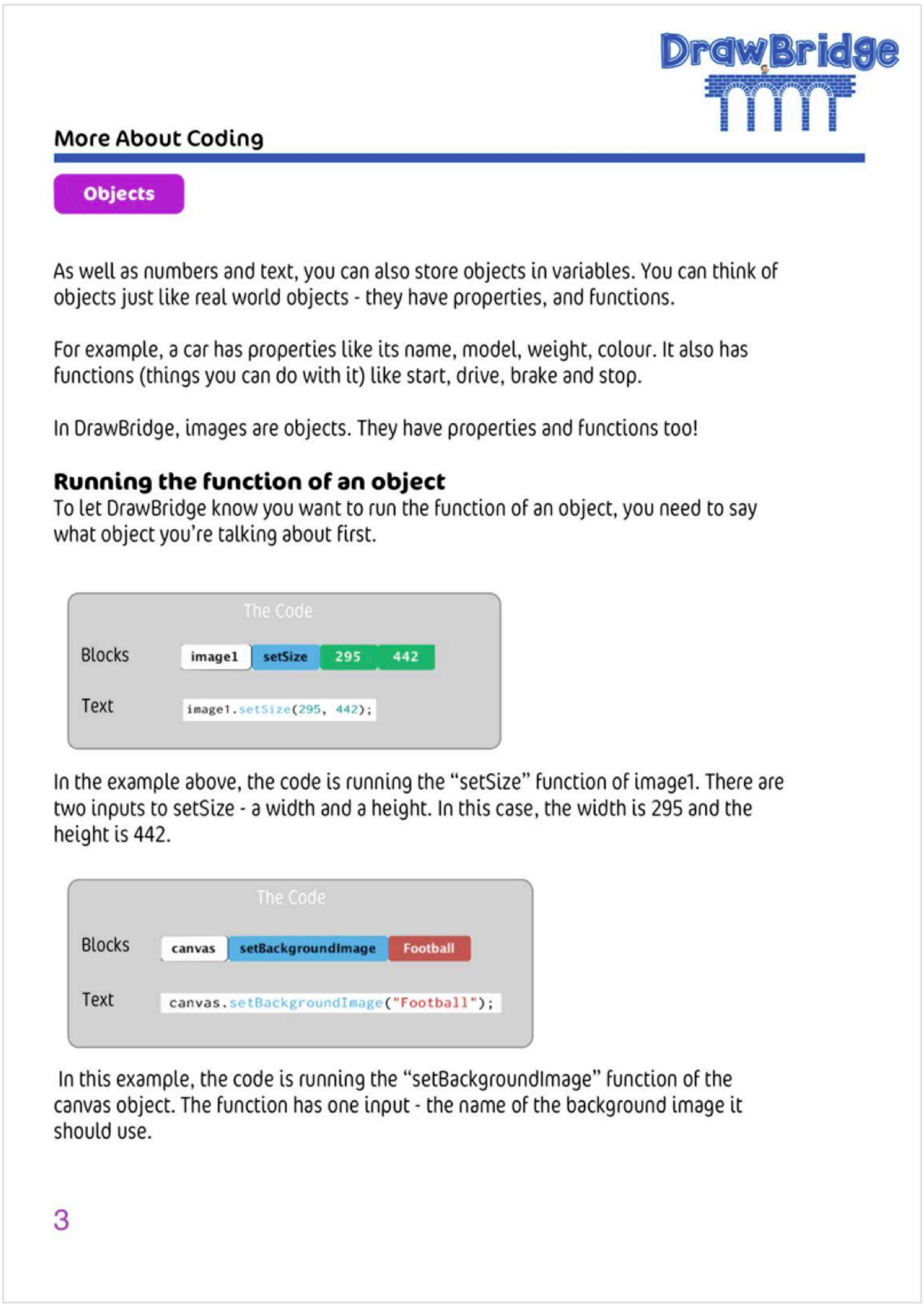
In DrawBridge some functions have been made for you. In the code example above, we create a variable called "image1" and give it the output coming from the function loadNextImage. There are no inputs, so nothing goes between the brackets.

What functions can I use in DrawBridge?

There are lots of functions in DrawBridge, here are a few of them:

- loadNextImage takes no input, and outputs an image.
- getRandomNumber takes a number, and outputs a random number
- beginAnimation takes no input, and gives no output
- commitAnimation takes a number (time in seconds) as input. It has no output





Objects

As well as numbers and text, you can also store objects in variables. You can think of objects just like real world objects - they have properties, and functions.

For example, a car has properties like its name, model, weight, colour. It also has functions (things you can do with it) like start, drive, brake and stop.

In DrawBridge, images are objects. They have properties and functions too!

Running the function of an object

To let DrawBridge know you want to run the function of an object, you need to say what object you're talking about first.

The Code

Blocks: image1 | setSize | 295 | 442

Text:

```
image1.setSize(295, 442);
```

In the example above, the code is running the "setSize" function of image1. There are two inputs to setSize - a width and a height. In this case, the width is 295 and the height is 442.

The Code

Blocks: canvas | setBackgroundImage | Football

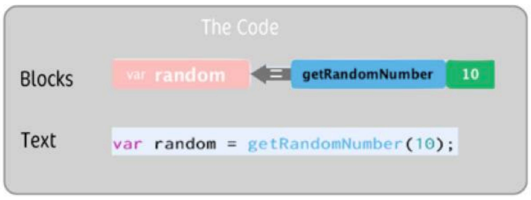
Text:

```
canvas.setBackgroundImage("Football");
```

In this example, the code is running the "setBackgroundImage" function of the canvas object. The function has one input - the name of the background image it should use.

Working with Random Numbers

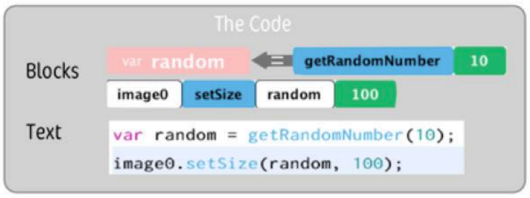
DrawBridge gives you a function to get random numbers! You can use this function to change the size, or location of your characters randomly.



```
var random = getRandomNumber(10);
```

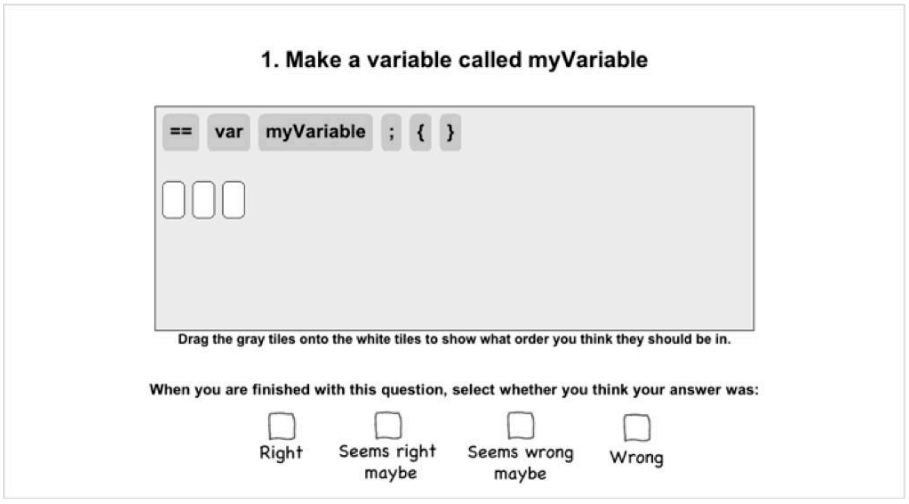
In the above code, we create a variable called "random" using the output of the function getRandomNumber with the input of 10. Using 10 will give us a number between 0 and 10. You can use a bigger input to get bigger numbers (e.g. 1000).

We can use the number we got to set the size or location of one of our images:



```
var random = getRandomNumber(10);
image0.setSize(random, 100);
```

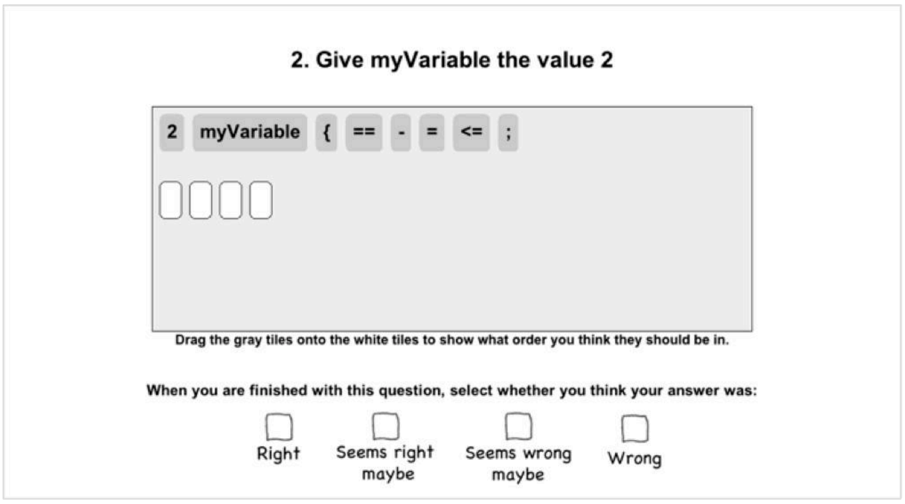
Pre/Post-Test



1. Make a variable called myVariable

Drag the gray tiles onto the white tiles to show what order you think they should be in.

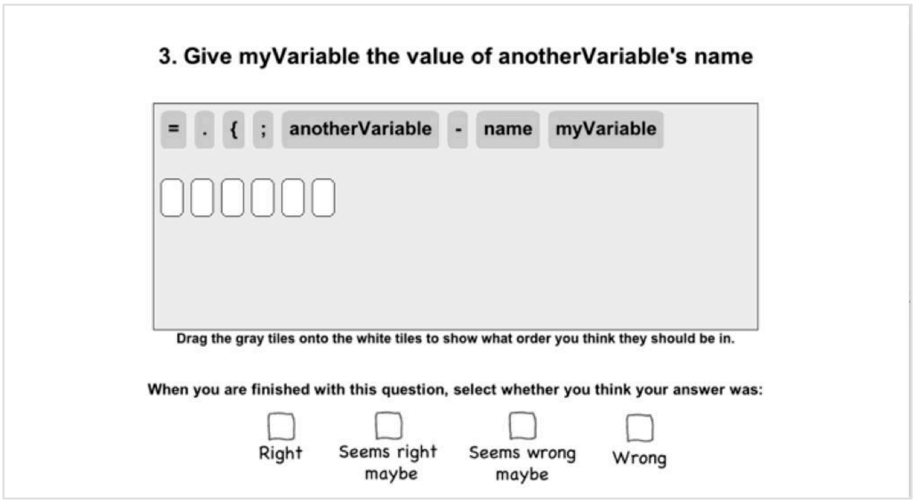
When you are finished with this question, select whether you think your answer was:



2. Give myVariable the value 2

Drag the gray tiles onto the white tiles to show what order you think they should be in.

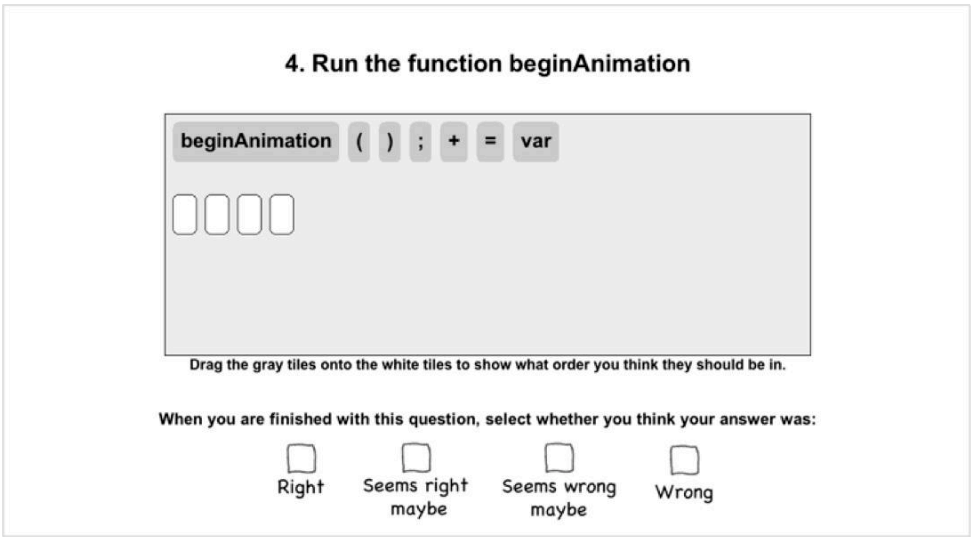
When you are finished with this question, select whether you think your answer was:



3. Give myVariable the value of anotherVariable's name

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:



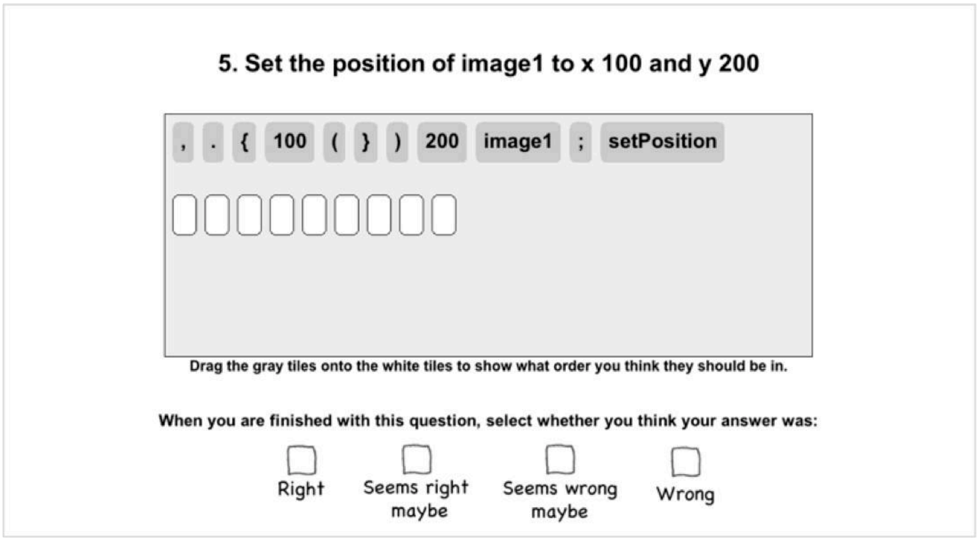
4. Run the function beginAnimation

beginAnimation ( ) ; + = var

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong



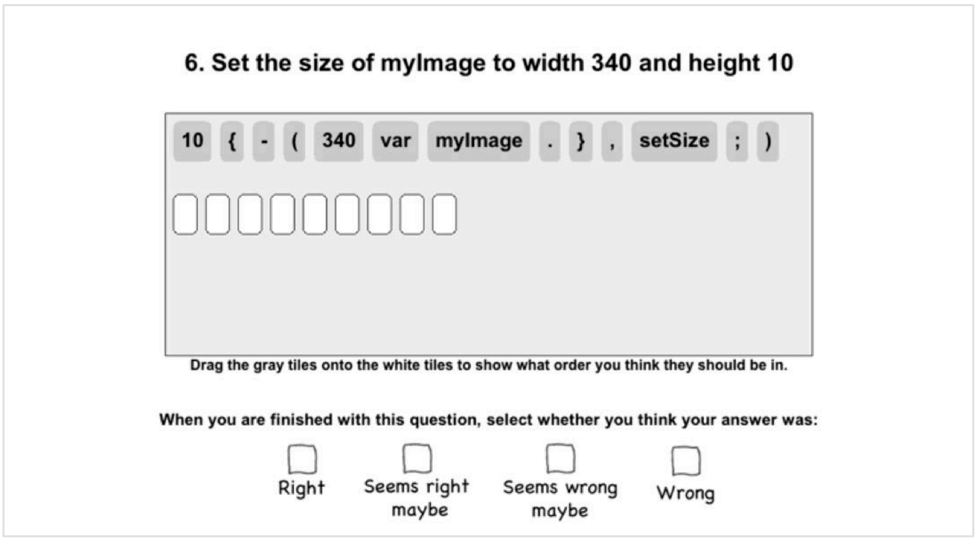
5. Set the position of image1 to x 100 and y 200

, . { 100 (}) 200 image1 ; setPosition

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong



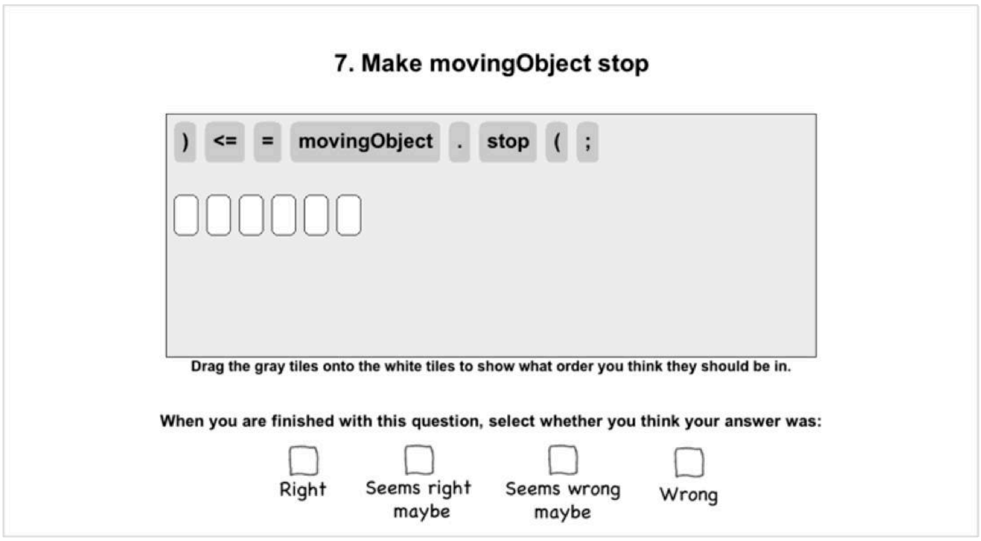
6. Set the size of myImage to width 340 and height 10

10 { - (340 var myImage . } , setSize ;)

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong



7. Make movingObject stop

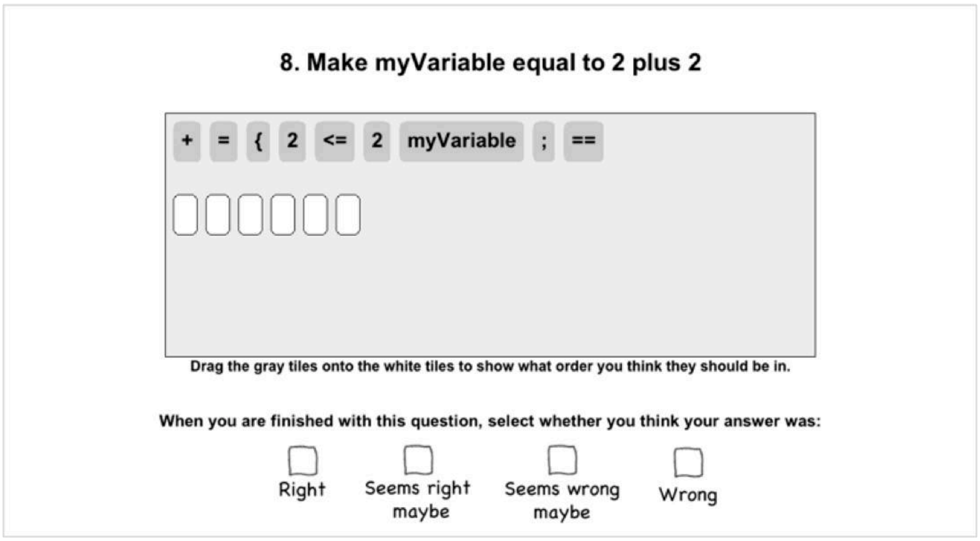
`)` `<=` `=` `movingObject` `.` `stop` `(` `;`

☐ ☐ ☐ ☐ ☐ ☐

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong



8. Make myVariable equal to 2 plus 2

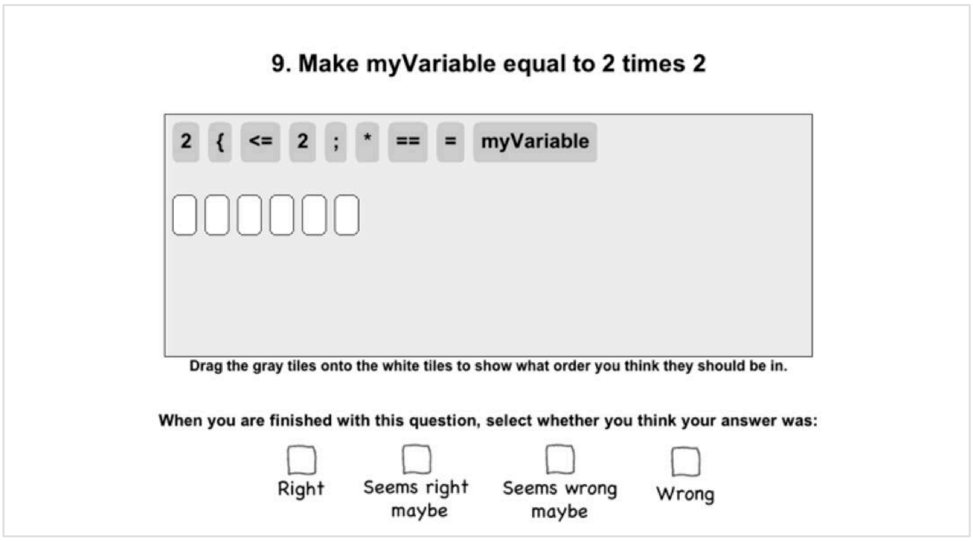
`+` `=` `{` `2` `<=` `2` `myVariable` `;` `==`

☐ ☐ ☐ ☐ ☐ ☐

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong



9. Make myVariable equal to 2 times 2

`2` `{` `<=` `2` `;` `*` `==` `=` `myVariable`

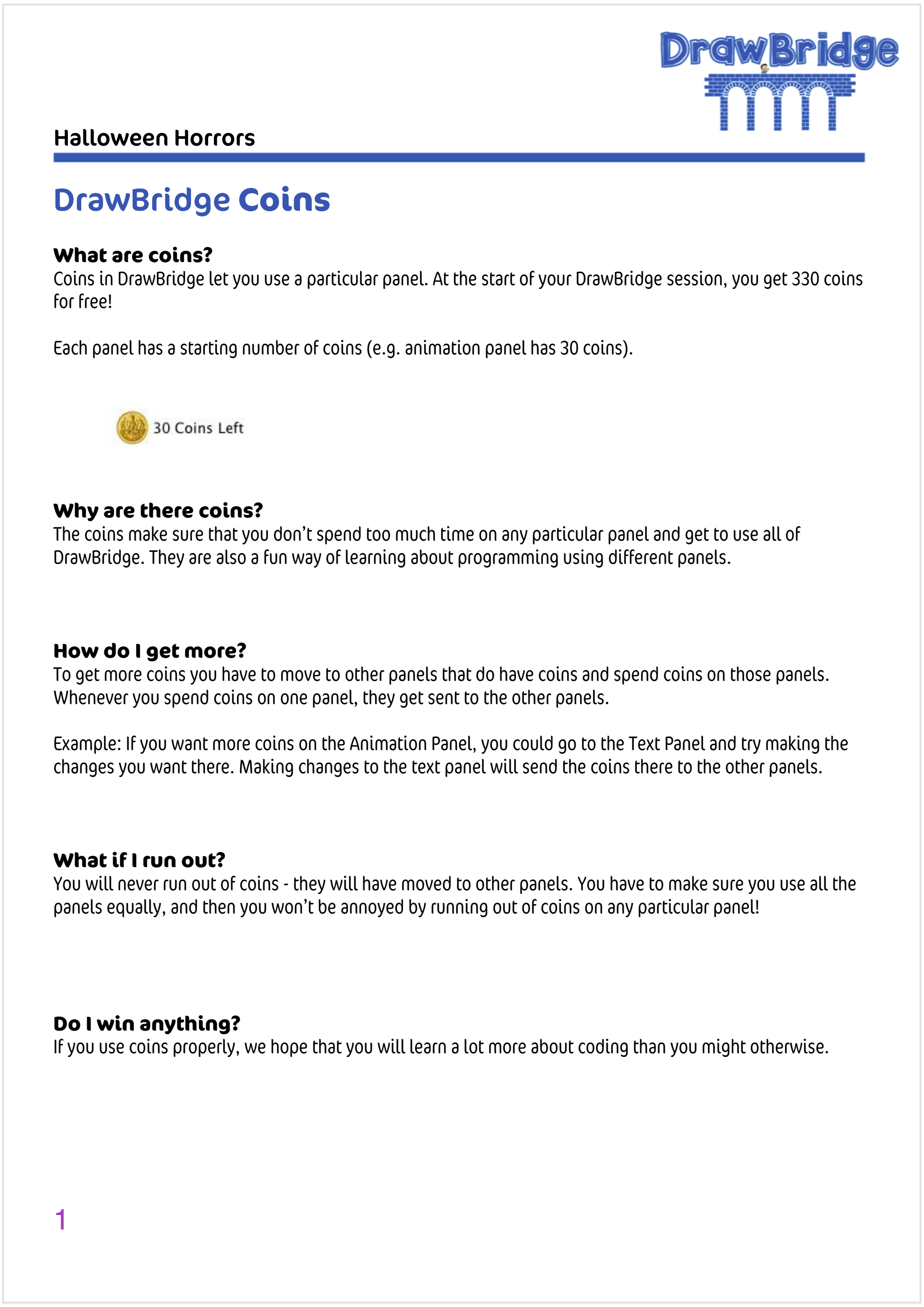
☐ ☐ ☐ ☐ ☐ ☐

Drag the gray tiles onto the white tiles to show what order you think they should be in.

When you are finished with this question, select whether you think your answer was:

☐ Right ☐ Seems right maybe ☐ Seems wrong maybe ☐ Wrong

Coins



Halloween Horrors

DrawBridge Coins

What are coins?
Coins in DrawBridge let you use a particular panel. At the start of your DrawBridge session, you get 330 coins for free!

Each panel has a starting number of coins (e.g. animation panel has 30 coins).

30 Coins Left

Why are there coins?
The coins make sure that you don't spend too much time on any particular panel and get to use all of DrawBridge. They are also a fun way of learning about programming using different panels.

How do I get more?
To get more coins you have to move to other panels that do have coins and spend coins on those panels. Whenever you spend coins on one panel, they get sent to the other panels.

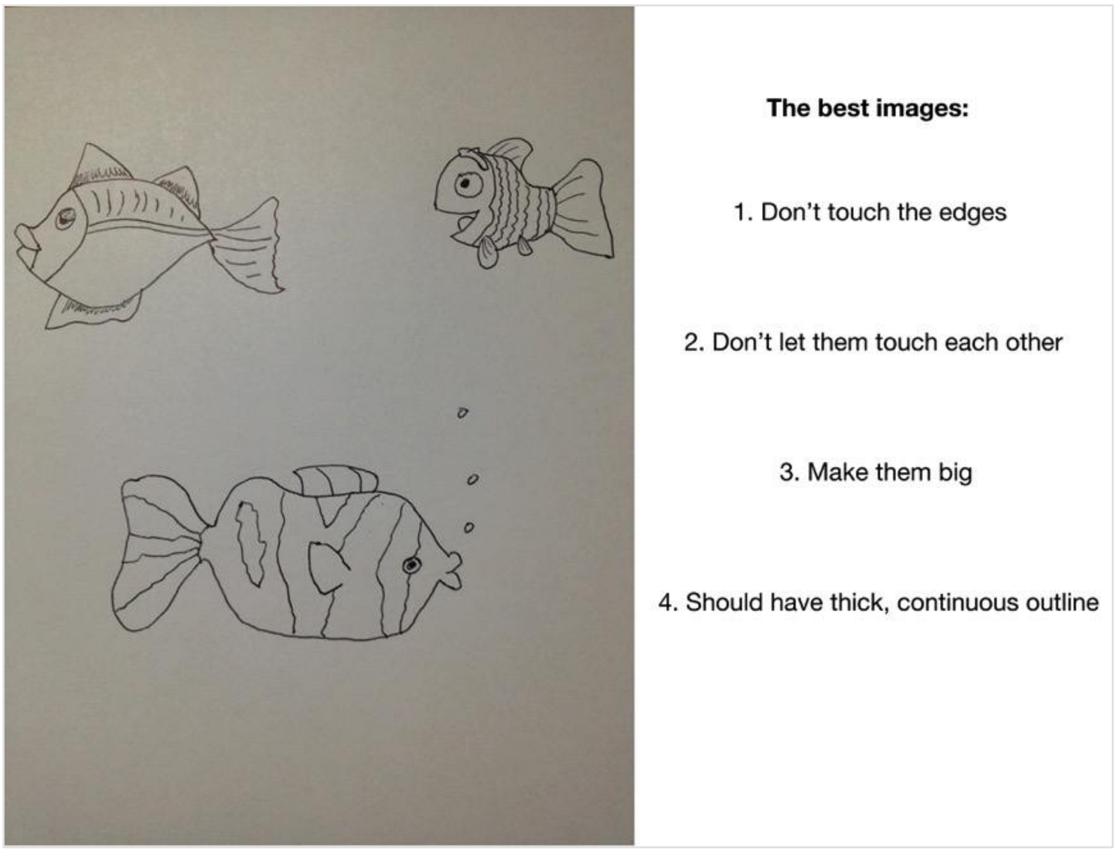
Example: If you want more coins on the Animation Panel, you could go to the Text Panel and try making the changes you want there. Making changes to the text panel will send the coins there to the other panels.

What if I run out?
You will never run out of coins - they will have moved to other panels. You have to make sure you use all the panels equally, and then you won't be annoyed by running out of coins on any particular panel!

Do I win anything?
If you use coins properly, we hope that you will learn a lot more about coding than you might otherwise.

1

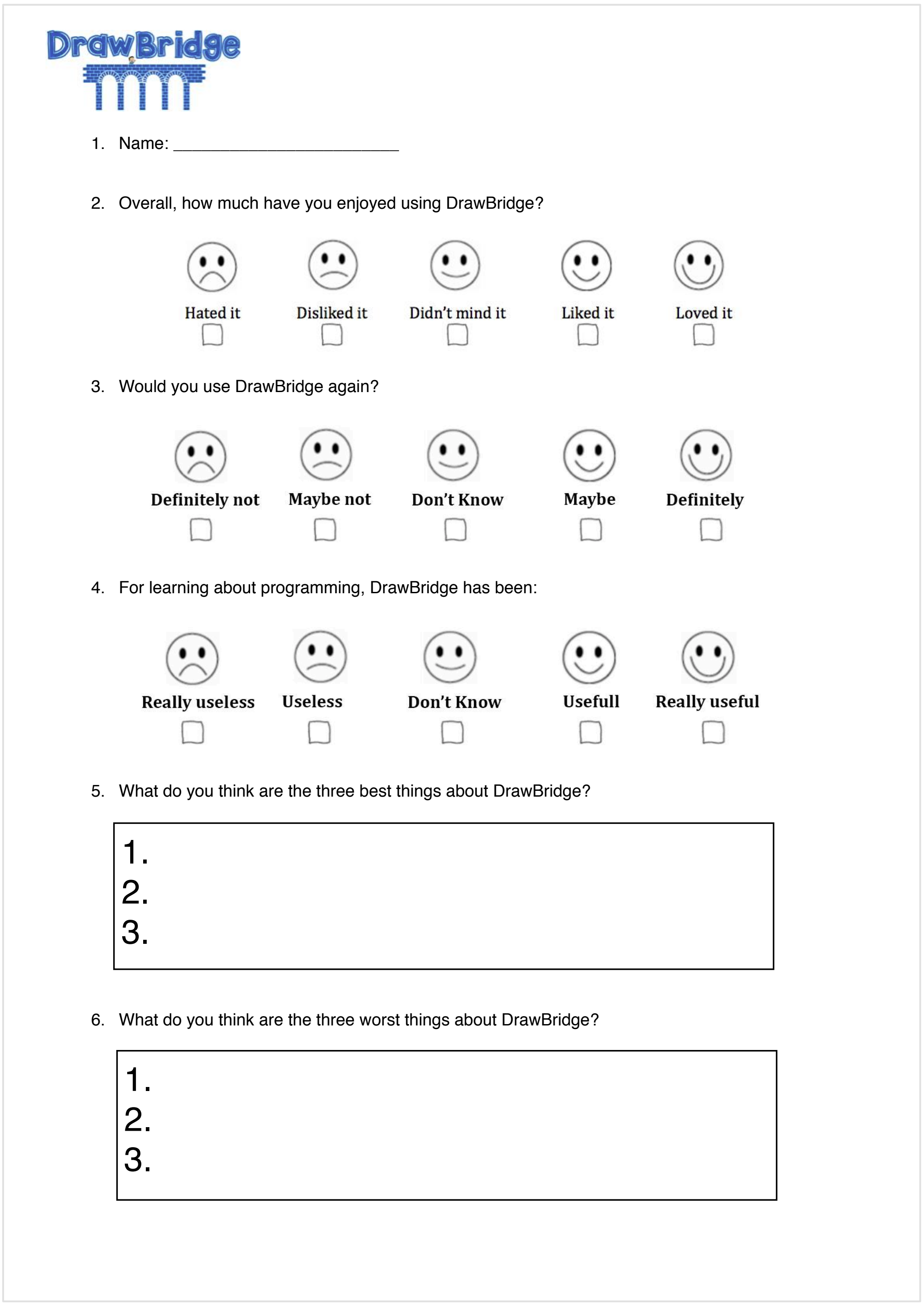
Drawing Rules



The best images:

1. Don't touch the edges
2. Don't let them touch each other
3. Make them big
4. Should have thick, continuous outline

Post-Questionnaire



DrawBridge

1. Name: ______________________

2. Overall, how much have you enjoyed using DrawBridge?

Hated it	Disliked it	Didn't mind it	Liked it	Loved it
☐	☐	☐	☐	☐

3. Would you use DrawBridge again?

Definitely not	**Maybe not**	**Don't Know**	**Maybe**	**Definitely**
☐	☐	☐	☐	☐

4. For learning about programming, DrawBridge has been:

Really useless	**Useless**	**Don't Know**	**Usefull**	**Really useful**
☐	☐	☐	☐	☐

5. What do you think are the three best things about DrawBridge?

1.
2.
3.

6. What do you think are the three worst things about DrawBridge?

1.
2.
3.

7. How could we improve DrawBridge?

8. How much did you like doing the quizzes before and after using DrawBridge?

Hated it	Disliked it	Didn't mind it	Liked it	Loved it
☐	☐	☐	☐	☐

8. How much did you enjoy using your own characters?

Hated it	Disliked it	Didn't mind it	Liked it	Loved it
☐	☐	☐	☐	☐

9. DrawBridge has made your interest in programming:

A lot less	**Less**	**Indifferent**	**More**	**A lot more**
☐	☐	☐	☐	☐