Register Machine Simulator

Jean Power

2106494

Supervisor: Dr Simon Rogers

A dissertation submitted in part fulfilment

of the requirement of the Degree of

Master of Science at the University of Glasgow

September 2014

Abstract

The aim of this project was the deployment of RodRego, a register machine simulator, to a modern platform. Understanding the functionality of a computer at a base level is an exercise in logic, and register machines echo basic CPU functionality. Creating programs using an appropriately designed register machine simulator conveys the concept of CPU functionality and develops problem solving skills.

Research pointed to the importance of interface design when aiming a product at naïve users. To ensure the best interface was developed, and to assess the learning outcomes, multiple user evaluations were performed. These indicated the developed application performed well; the design was accessible to novice users, conveyed the concept of a register machine and, consequentially, CPU functionality.

Educational Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Signature:

Contents

Introduction

In 1936, Turing described a hypothetical computing machine with infinite memory and a scanner that moved along this memory, an unending strip of tape holding symbols, usually presented as binary (Turing, 1936). The machine follows a predefined list of instructions, reading a symbol from memory and performing an action, dependent on the symbol – erase, or write, then move the scanner right or left. Depending on the symbol, the machine may branch to another instruction, or move to the next in the list. Given enough time, a Turing machine can compute any function, in part due to the conditional branching ability. In this way, it can simulate any real-life computing device.

The principles behind the Turing machine underlie modern computing, and are fundamental for understanding the CPU functionality. However, it cannot be fully simulated, due to the requirement for infinite memory. This gave rise to the register machine, defined by Wang in 1957. These are similar to Turing machines with simple instruction sets of arithmetic and conditional operations, but have finite memory, lending themselves well to being simulated on modern computers.

Importantly, register machine functionality also corresponds at a base level to how a CPU works; Dennett states that "Turing Machine[s] and…Von Neumann Machine[s] … are 'just' register machines with speed ups and more memory" (2008).

RodRego is a counter register machine simulator, originally developed in 1986, to demonstrate the computational capabilities of a programming language containing three instructions, and an allocation of ten data registers (Dennett, no date, Dennett 2008). The instruction set of increment, decrement/branch, and end can combine in surprisingly complex ways, laying a foundation of understanding for CPU functionality. Unfortunately, despite some strong positives, RodRego is no longer fit for purpose, as it does not run on many modern operating systems. These types of educational tools are core to understanding how computers work, but RodRego comes equipped with no help file. It is therefore unintuitive, and inaccessible for users without prior knowledge of the register machine concept. The underlying theories, although logical, can be difficult to understand, and the interface does not reflect this.

This project aims to transfer RodRego to a more accessible medium, keeping the positive aspects and improving on the negative. Importantly, the user interface and tutorials will be focused on, to aid in accessibility, and conceptual understanding. Therefore, the user evaluation will be key to the success of this project.

First, the problem statement will be clearly defined to provide a rational path of research. The background survey follows on from this, structuring an approach to develop solutions to these problems. This includes validating solutions to similar obstacles. Finally, a delineation of the aims of this project, ensuring satisfaction of the problem statement, and the proposed work approach will be defined.

Statement of problem

Educational tools like RodRego are essential; computers impact almost every aspect of human life, and only a handful of people fully understand their inner workings and logic. The concept of a register machine is fundamental to understanding how modern computers work, without being overwhelming in its complexity. If a simulator is appropriately designed, developing simple programs will help users learn basic computing theories, as well as promoting the crucial skills of troubleshooting and logic.

Appropriate design and feedback is important as, together, they can create a positive learning experience. It has been shown that these experiences shape users' self-image, attitude towards computers, and perceived ability. Impacting this provides motivation to explore and discover further computing techniques and processes (Schulte & Knobelsdorf, 2007).

RodRego V1.3 was developed in 2001 using Visual Basic 5 (Dennet, no date). Visual Basic runs on the Windows .NET framework, making it impossible to run on non-Windows machines without additional software. Also, it does not run on some modern versions of the Windows OS. There are other major flaws; there is no help documentation, the interface is unintuitive, and instructions must be typed in. All of these are stumbling blocks for naïve users. It was clearly intended that RodRego would be used with prior knowledge of register machine functionality.

This project seeks to create an educational tool, using the concept of the register machine that RodRego is based on, with intended users having no prior knowledge of the theory of register machines. It will connect this to CPU functionality, with a goal of improving the user's understanding of how computers work.

Background Survey

In the development of this educational tool, many aspects affect the design and implementation. Primarily, the concept of a register machine and its functions must be preserved, and the theory of this conveyed to the user. This is the central goal of the project; it impacts the user interface, as well as the choice of technology, and the avenues of research in the development of this proposal.

Further to this, it is important to consider when a user will interact with the tool, and how the context of the situation will impact on learning outcomes. The aptitude level of the average user must be taken into account to ensure the information is appropriate, it can successfully convey the theory of a register machine, and how this relates to CPU functionality.

What follows is a brief overview of "The Secrets of Computer Power Revealed", the paper which RodRego is based on, and a literature review which explores the above facets and their impact on this project.

The Secrets of Computer Power Revealed

Dennett puts forth that, even though it is universally known that a Turing-complete machine (one which can simulate a Turing machine) can compute any algorithmic function, it is not universally understood. To improve this understanding, he describes a register machine with an instruction set of increment, decrement/branch, and end, described in Table 1.

Table 1 – Dennett Instruction Set

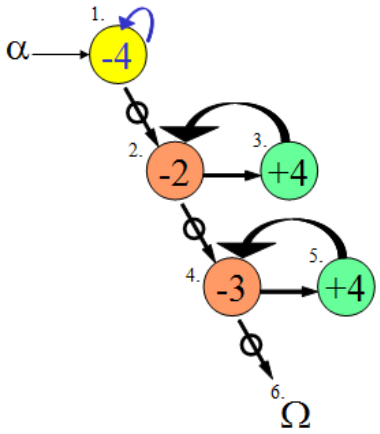
Instruction	Example	Explanation
Increment (INC)	1. INC, 1, 2	Increment Register 1, move to Step 2.
Decrement/Branch (DEB)	2. DEB, 2, 1, 3	Decrement Register 2, move to Step 1, unless Register 2 contains zero; then branch to Step 3 instead
End (END)	3. END	End

He creates an Add program, taking the data from Register 2 and Register 3, placing this data in Register 4. This is demonstrated in Figure 1 as both a flow graph, and in Dennett's Register Assembly Programming (RAP) language. Each circle in the flow graph is an increment, or decrement, the register indicated by the number. The program starts at alpha (α) and finishes with omega (Ω), following the loops and, when the register contains zero, the branches. A branch is indicated by a circle on the arrow.

The first step clears Register 4 by repeatedly decrementing until zero, when the program branches to the second step. A loop of decrementing Register 2 and incrementing Register 4 follows, with a similar loop for Register 3. Finally, the program branches to end when Register 3 contains zero. This leaves the contents of Register 2 and Register 3 in Register 4.

Dennett builds on this simple example, showing how moving, copying, adding, multiplying, and so on, can be completed. Division, for example, is achieved by a loop decrementing the divisor and dividend, keeping track of each full decrement, restoring the divisor and repeating the process.

Figure 1 – Add Program



Step	Ins	Register	GoTo	BranchTo
1.	DEB	4	1	2
2.	DEB	2	3	4
3.	INC	4	2	
4.	DEB	3	5	6
5.	INC	4	4	
6.	END			

With conditional branching, this language contains the power of subroutines; Dennett asserts a program can be written to fully simulate a calculator using this hypothetical register machine and instruction set.

A series of subroutines would be required, one for each mathematical computation. The user would enter the operation in the first register – perhaps 0 for ADD or 2 for MULTIPLY – followed by the operands, their signs indicated by a flag in the succeeding registers. The register machine would decrement the first register and the first in a series of stored numbers, the match of the operation numbers. When the first register reached zero, the program would branch to the stored number; if this is not also zero, it is the incorrect operation. The program then branches to a helper subroutine, restoring the first register and the stored number, before repeating the process with the next stored number. At some point, these would match, and the program would branch to the start of the correct subroutine, performing a computation with the operands. Error flags could also be held for issues like attempting to divide by zero. The only limit to the number and complexity of the operations is the amount of memory locations the register machine has been assigned.

Dennett reminds the reader that, in this example, the data in the registers is being used for different purposes; a number, an error flag, signs, and an operation. It is the programmer who gives meaning to these numbers to interpret them, a core concept in CPU functionality. The set of subroutines, accessed by input from the user, is the basis of a stored program computer.

The register machine works with short, simple instructions and relatively small amounts of memory. Despite the fact that today's computers can work with 32/64bit instructions and billions of memory locations the basis of reading, executing and writing is the same as in a register machine.

Learners are often presented with basic concepts and principles, prior to being taught a subject in depth. This method develops baseline knowledge which supports more complex concepts. Dennett uses the register machine to improve understanding of the Turing Machine and Turing-completeness. The tool to be developed will use this same tactic, with the goal of improving understanding of CPU functionality.

Learning Environment

It has long been suggested that learning outcomes are influenced and constrained by the context of the learning experience (Brown *et al.* 1989). This is similar to computer use, defined and shaped by social and cultural situations (Sutherland, 2000). Both of these are important to consider when developing educational applications, as formal, non-formal and informal learning environments have differing impacts on cognition. Therefore, the 'Where' and 'How' of this tool is critical to consider, as it has consequences on learning.

Separation of learning environments is difficult, as boundaries are blurred by each containing attributes of all environments, but a broad definition can be applied by reviewing the learning itself. Formal learning is compulsory, intentional study leading to accreditation, with the learner making choices within a predetermined, prescribed syllabus. It is most closely associated with traditional classroom education. Non-formal learning is similar; the learner takes part in structured schooling, but they have chosen to partake, and the learning is adaptive to their needs. Swimming lessons or field trips can be considered non-formal learning environments. Finally, informal learning comes about spontaneously, in an unstructured way in everyday life, within a familial or social context. The learning is often led by the learner, and is intrinsically linked with their own motivations (Mocker & Spear, 1982, Vadeboncoeur *et al.* 2006, Eshach, 2007).

There are benefits to non-formal and informal learning. They are effective, with long-term impacts, and improved learning outcomes over formal learning alone. It has been suggested that the majority of science learning takes place outside the classroom, with real world experiences increasing scientific reasoning and, in turn, learning outcomes in the formal environment (Gerber *et al.* 2001). Students who took part in field trips significantly outperformed classmates on scientific attitude tests, a measure of the qualities of mind required for proper scientific investigation (Harvey, 1951). A review of six studies by Falk (1983) found that significant long-term learning occurs on field trips, and this was confirmed in further studies by Lam-Kan (1985) and Finson & Enochs (1987) who corroborated the link between participation, and improvements in scientific concept attainment and attitude towards science. After-school programs, a non-formal learning environment, have also been shown to improve mathematics test results, attitudes towards formal school structure, and attendance (Reisner *et al*. 2004).

Taking this into account, this tool will be developed for use in an informal environment. It is hoped that the user will choose to partake and will interact with it informally, rather than it be disseminated for use in a structured, formal environment. It is believed that this will lead to improved cognitive outcomes.

PC versus Mobile Phones

Mobiles devices are ubiquitous; the number of smartphones doubled, and data traffic increased almost 300% in only two years (CTIA, 2012). This has affected how we communicate, how we entertain ourselves, and how we learn. There are strong motivations to participate in mobile learning. It is believed this is due to a sense of control, and ownership that users feel over a device, as well as the ability to access it in any context with continuity of material between these contexts (Issroff *et al.* 2007).

Other studies support these outcomes. A study in 2005 found that an overwhelming majority of students preferred accessing e-mail on their mobile phones (Thornton & Houser, 2005). This same study showed that students who received English lesson material over mobile e-mail performed better, in comparison to those who studied it online. Being able to access the learning material everywhere was seen as important, as the students often studied it on their commute home.

This is not an isolated effect. Over a three year period, despite taking additional time to complete, increasing numbers of students opted to use a mobile device to access English vocabulary lessons (Stockwell, 2010). The perception of mobile learning is changing, as it allows users to add productive tasks to otherwise wasted time, like commuting. The ability to integrate these learning activities into everyday life, rather than it being a separate task, gave learners freedom in both time and space. This type of independent learning, in non-formal environments, links in with the previous discussions of improved cognitive outcomes in these settings.

In particular, lower aptitude users had improved test scores when they received text messages, suggesting topics and quizzes that could be accessed through their mobile device. This was in combination with an online learning system, accessible to all students. Users, who received messages through their mobile device, rather than on the website itself, spent twice as long on the learning system and produced improved test scores (Chen *et al.* 2008).

Repetition is the cornerstone of learning, and the spacing of this is particularly important; performance is improved if the same amount of repetition is distributed across time, instead of massed in a single session. This effect has been researched on over 300 separate occasions, and improvements in outcome have been shown in many disparate areas; physical activities like typing, contextual syntax of English grammar, and fact based learning of history (Baddeley & Longman, 1978, Bird, 2010, Cepeda *et al.* 2006, Carpenter *et al.* 2009). It has also been shown to boost test scores in mathematical tasks (Rohrer & Taylor, 2006). Increasing access to learning material should take advantage of this spacing effect, improving learning outcomes.

This all suggests implementation for a mobile device, but a PC based system is still a valid option, as mobile learning will rarely afford a fully immersive experience. A few minutes interacting with a mobile device whilst waiting for a bus could simply never impart as much information as a PC application; large passages of text are difficult to view on small screens, and mobile data connection problems can hamper video viewing and internet connectivity. When learning a language, students found authentic materials, such as novels and TV shows

important, as well as scaffolding – hearing the language as well as seeing it written down (Fallahkhair *et al.* 2004). These are difficult to impart solely through the mobile medium.

However, choosing to deploy as a mobile application is sound. Most importantly, the motivational factors should increase the amount of user interaction. It provides more access opportunities, taking advantage of the spacing effect. It could also improve as the users are naïve to the concept of a register machine. Due to the limited amount of information to be conveyed, the system does not need to be fully immersive. The implementation itself will benefit from the arguably more flexible, interactive nature of a mobile device, with intuitive gesture capture and portability. This choice also reflects the increasing user preference for mobile learning, and should improve cognitive outcomes by reducing the formality of the experience. Finally, there is strong evidence to suggest that users are more likely to download unbranded, unknown applications to their mobile phones than onto their PC or laptop (Chin *et al.* 2012). This would increase both the number and range of users.

Teaching Complex Concepts

Complex concepts, such as the inner workings of the CPU, have specific requirements for teaching the theories fully and correctly. The medium and context of the message and the level of engaged thinking play a major role in imparting this information properly.

In teaching mathematics to children, it is important to use numbers in a meaningful way. If it is only taught in a computational sense, with one right answer and no real world connection, the knowledge can only be applied directly. Encouraging a class to connect normal procedural knowledge with intuitive knowledge, through debate, led a majority of students to deduce mathematically sound, general observations in line with mathematical laws (Lampert, 1990). In this way, complicated theories were organically discovered and understood. Introducing people to register machine simulators, ensuring they have the procedural knowledge, and allowing them to problem solve within this should allow for this natural progression of thought. This developed conceptual knowledge is then relevant for understanding CPU functionality.

Visualising examples, ensuring symbolic expressions are linked with real world phenomena, allows students to develop a more complete and consistent mental model of concepts (Brasell, 1987). This can be assisted by technology; chemistry students using software that had synchronised animated, graphical, symbolic and video representations of virtual experiments, controlled by the student, decreased misconceptions and increased accuracy (Russell *et al.* 1997). Importantly, the students enjoyed these lessons; in other studies, combining audio, animation and video, the tasks were considered fun, evocative and transparent (Thornton & Houser, 2005). The students also learned effectively. Even though the register machine representation is symbolic, combining it with the real world function of computing results will lead to a better overall understanding of how a CPU works.

Humans are natural storytellers and tapping into this enhances learning outcomes; teaching complex medical concepts through storytelling gave a statistically better result over classroom teaching (Vali, 2007). This links in with overlying a meaning to improve learning,

when it comes to teaching difficult theories, and implies that story based metaphors are particularly effective.

An important use of technology is gaming, with it being the most common computer operation amongst children (Mumtaz, 2001). A feeling of control and being challenged increase involvement with educational games, leading to active learning; this, in turn, improves learning outcomes (Wishart, 1990). Interactive media, like games and simulators, significantly outperform traditional teaching methods, especially where the learner controls their own navigation through the system, personalising their experience (Vogel *et al.* 2006). Therefore, even with a difficult concept like CPU functionality, a balanced combination of educational content and a challenging game should appeal to users, and lead to improved cognitive outcomes, over classroom teaching.

Impact of Design

It is clear that mobile learning is an important tool, and one that will be relied upon as education moves outside the classroom. Massive open online courses (MOOC) continue to increase in size and number and, as mentioned previously, integrating technology into education leads to better results (Pappano, 2012). The next step is to convey the information appropriately. To ensure this, the content and design of the tool is critical. If the interface is not accessible and user friendly, if the content does not capture and transmit the information, the user will not gain anything, and the tool has failed.

Below is a table adapted from Najjar (1998), outlining the principles of educational multimedia user interface design. These design principles suggest ways in which the tools can be developed to improve learning.

Table 1 – Principles of Educational Multimedia User Interface Design, adapted from Najjar (1998)

Characteristics of the materials
Use the medium which best communicates the information
Use multimedia in a supportive, not decorative way
Present multimedia synchronously
Use elaborative media
Make the user interface interactive

Characteristics of the learner
Use multimedia with naive and lower aptitude learners
Present educational multimedia to motivated learners
To avoid developmental effects, use educational multimedia with adults and older children

Characteristics of the learning task
Use multimedia to focus a learner's attention
Encourage learners to actively process the information

Characteristics of the test of learning
Match the type of information tested to the type of information learned

With regards to the design of this simulator, some of these principles are particularly relevant. In general, pictures are far superior to text and motion based information should be displayed as a video or animation to fully convey the concept. The display must only contain relevant information; unrelated media distracts the user and decreases learning. Elaborating, for example, text with other media has a particularly positive effect on learning; multimedia encourages the user to engage multiple cognitive channels for processing the material. However, it is important to not overload input channels; text with accompanied audio is redundant and reduces learning (Kalyuga *et al*. 1999). In a choice between audio and text, Clark & Mayer (2011) suggest using text when a learner is naïve and has the time to process both pictures and text. Interaction with the material pushes users to actively process information, as long as it is engaging; they learn faster, have a better attitude to the material, and retain the information for longer.

This type of multimedia experience has better outcomes with naïve users. It is most effective when the user has no prior experience in the domain. However, they must be intrinsically motivated to take part – naturally, motivated learners will learn more than unmotivated. This motivation can be increased by relating the content to familiar, analogies and situations, in an informal manner, and providing immediate positive feedback. However, informality must be carefully handled, as humour distracts from the core concepts that are being conveyed.

Another aspect that will impact the design is the choice of colour; to increase accessibility, the colours should be chosen to prevent confusion for the colour blind. Without this, the 8% of men who have some degree of colour blindness would be excluded from using the application (Colour Blind Awareness, no date). There are websites which suggest the best colours to use and others to increase the red/green contrast of images; this improves the ability of the colour blind to discriminate between colour variations. (Vischeck, no date, Safe Web Colours, no date)

Studies have been completed on design, and the importance of good design for learning. Having unity, a focal point and balance are important for good design. Unity can be achieved by having elements grouped; the space between each object should be less than the size of the object. The centre of interest should be clear to the user, to grab and hold their attention. This can be created by making an element different or placing it outside a group. Finally, a balance on both the x and y axis of a screen promotes comfort in users. By changing a design to adhere to, or not, these three principles, researchers altered completion rates and time to completion in e-learning (Szabo & Kanuka, 1999). When there is no instructor, these aspects are important for the success of learning outcomes. Luckily, it did not affect achievement scores, which is encouraging information if any incorrect design decisions are made.

Evaluation of current solutions

An essential aspect of project design is evaluation of existing applications to ensure that bad design decisions are not repeated, and positive aspects are incorporated. To cover both possible outcomes of the project, a mobile and PC based version of a register emulator were chosen. It is difficult to objectively define what constitutes a usable application, as each user

will have specific requirements and different aptitude levels. To ensure consistency in evaluation, usability heuristics were used as a basis for review.

The usability heuristics chosen were those put forth by Neilson (1995):

Visibility of system status: System operation feedback should be appropriate to the user, and be displayed at appropriate times.
Match between system and the real world: The system should use natural language.
User control and freedom: Undo and redo should be simple, and easy to complete.
Consistency and standards: Follow conventions for the platform, ensure consistency within the application.
Error prevention: Reduce possibility of user errors.
Recognition rather than recall: Instructions, possible actions, current data should be visible and easily recognisable.
Flexibility and efficiency of use: Tailor experience for novice and advanced users, by allowing advanced users to save frequency actions or adjust settings, for example.
Aesthetic and minimalist design: The screen should only contain relevant information.
Help users recognise, diagnose, and recover from errors: Error messages should have natural language, and suggest a solution.
Help and documentation: Help documentation should be easy to find, clear, short and list concrete steps to complete a task.

In addition, for the mobile application, an additional heuristic of **context of use** was taken into account, as this was recently recognised as playing a critical role in usability studies for mobile device software (Harrison *et al.* 2013). And, due to the educational nature of this project, the outcome of application use was documented; **was the educational concept clear**?

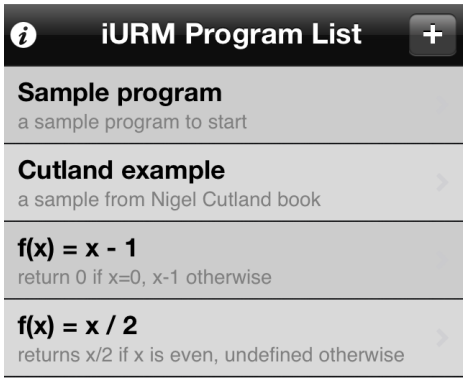
A user persona was developed to assist, as this further deepens the understanding of user requirements, whilst limiting their scope, during evaluation. The starting point for the user persona was the assumption that the application is not intended as an introduction to computers, but as a tool to enhance the understanding of the inner mechanics of a computer. An overview of the user persona developed follows.

The user will be comfortable with technology and using applications. They will find computers and mobile devices relatively easy and enjoyable to use, and understand they can be customised with software. The user will have experience with educational applications. The user will have no understanding of how a CPU works, and will look for a hands on application to develop an awareness of this concept. The user will have motivation to seek out the application and, after use, will expect to have grasped the concept. There is no specific age range for this average user.

IURM (Unlimited Register Machine) (De Bortoli, 2010)

Created by De Bortoli in 2010, this register machine has a limited instruction set of Z (zero), S (increment), T (transfer contents) and J (compare and jump). It was developed for iOS.

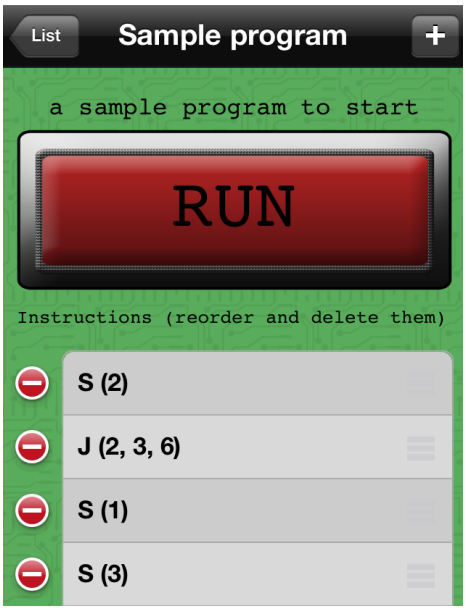
Figure 2 – Program Screen



The first screen is clearly labelled as the program list, and has an obvious 'Help' button, as well as an 'Add' button. Each program has a brief explanation. Between sessions, user saved programs can be stored.

However, there is no tutorial and only high level, sparse help. This is likely due to the intended audience already having experience with the theory of a register machine, as indicated by the reference Nigel Cutland book; this is a university level text.

Figure 3 – Edit program screen



Clicking on a program brings the user to a screen dominated by a 'Run' button. Again, a clear 'Add' button for instructions is visible, accompanied by a short explanation of what is expected of the user. Undoing or moving an instruction is simple through the platform consistent symbols for delete/move.

There are issues for novice users. The Jump instruction requires a parameter of the number instruction that is jumped to, but there is no indication of what number each instruction is.

Figure 4 - Add/Edit instruction screen

When adding an instruction, the correct number of textfields for arguments dynamically appears, reducing the possibility of errors. Cancel or save are both clear options.

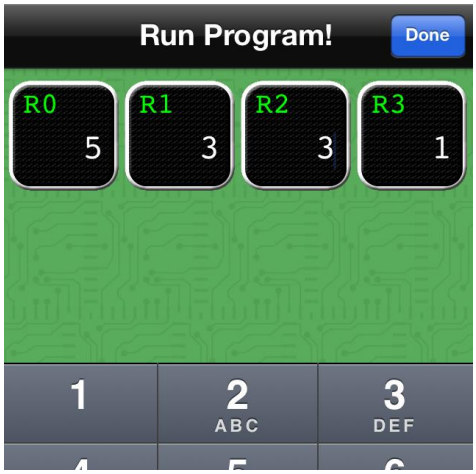
But, if an argument is not entered, it is defaulted to '0', instead of alerting the user to the issue, leading to unexpected effects. The user is required to remember what each instruction means, and how the arguments are ordered, as there is no link to the help text.

Figure 5 – Run Program screen

Clicking on 'Run', a 'Run Program' screen appears. This only displays registers affected by the program, reducing confusion. Any 5 digit number can be entered into the registers, giving flexibility. As each instruction is executed, it is highlighted on the scroller and on the PC (program counter) indicator. The register icons also flash when they are impacted by an instruction, and slide to the next register when they transfer data.

However, the program does not run automatically when 'Run' is clicked. The user must then 'Exec' the program, which can be easily confused with 'Begin' (a command to return to the first instruction. The ability to edit the registers and enter data is unclear, even though this is often necessary to see the effects of running a program.

Figure 6 – Done button issue



Finally, the 'Done' button returns the user to the instruction screen, which causes problems when parameters are being entered into the registers – users automatically select 'Done' when they have completed editing the registers, losing their edits. Even for users with experience of register machines, this screen is confusing.

In some ways, this solution is elegant. It does not remove itself with metaphors from what it is trying to teach. This ensures there is no confusion, and the message is clear. In many places, the interface is intuitive, using the platform specific conventions. The design is minimalist, with no unnecessary information and undoing is simple. Only allowing the correct number of arguments to be added to an instruction, and the instruction itself selected from a list, vastly reduces errors by the user. Simple animations as instructions are completed add to the experience of use and improve clarity, while showing the system status. The context in which this application will be used has been taken into account; an uncluttered, clear, platform specific design allows for successful use in environments where a user is not paying full attention.

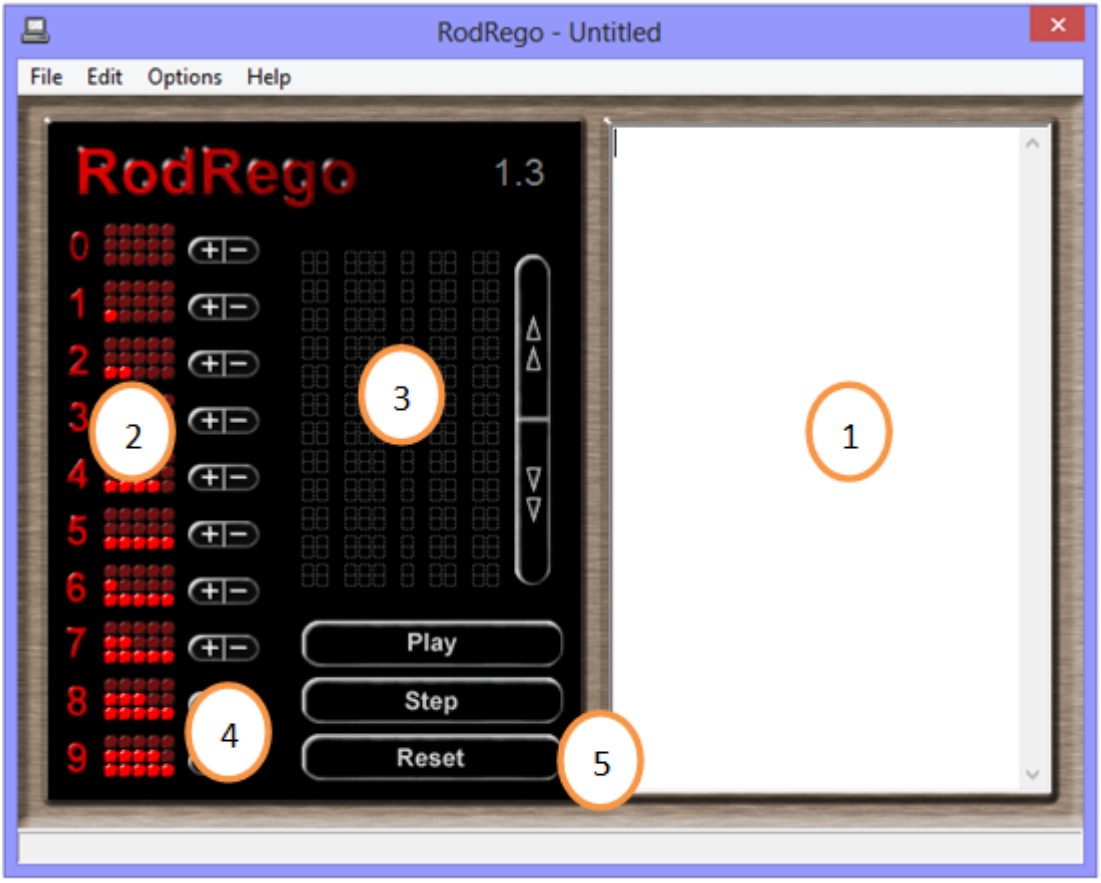
However, this does not carry across the entire application. Having an 'Exec' and a 'Begin' button is confusing. There is no feedback if an instruction does not contain the correct number of arguments. This is compounded by scant help text that is inaccessible from screens other than the main view. If a user was not already aware of how iURM should correspond to CPU function and computer memory, they would not develop this understanding from usage as this connection is not documented. This makes it inappropriate for a novice user.

RodRego V1.3 (Dennet, no date)

As previously described, RodRego is a PC based register machine simulator with an instruction set of INC (increment), DEB (decrement or branch) and END.

There is one GUI in RodRego; this evaluation will refer to specific elements, as numbered in Figure 7.

Figure 7 – RodRego V1.3

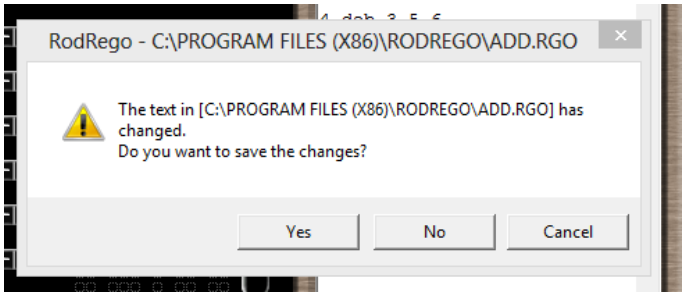


The interface for RodRego is minimalist, with no unnecessary media. It follows platform specific conventions, like Save/Load being present in the File menu, and many of the buttons have clearly associated actions.

Writing, Loading, Saving Programs

RodRego comes with example programs which can be loaded. Instructions can also be written or copy-pasted into (1). The user has the ability to save their creations. Within the program, each instruction has its own number, making it obvious where a DEB instruction will branch to (Fig. 10).

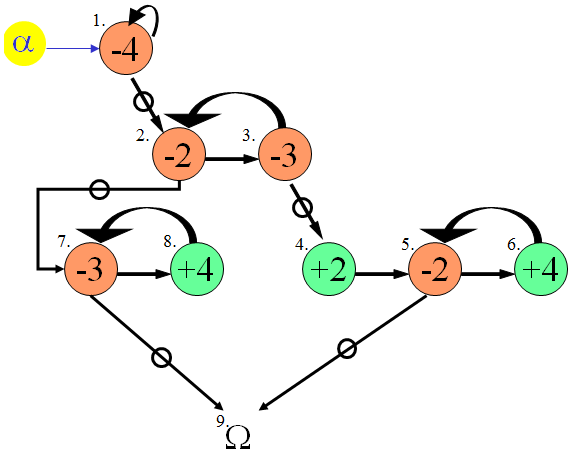
Figure 8 – Error Message



The system also warns the user to save their data if they attempt to exit without saving, reducing accidental data loss.

However, RodRego has no associated help or tutorial. Users must read "The Secrets of Computer Power Revealed" to understand the instruction set (Dennet, 2008). PowerPoint presentations from the website of program representations are aimed at users who already have an understanding of register machines and the flow graph notation used to denote a program.

Figure 9 - Flow graph of subtraction of 2 registers (Dennet, 2008).



For example, the flow graph in Figure 9 shows a subtraction, taking two scenarios into account – where the data in Register 2 is greater than or less than in Register 3.

Figure 10 – Entered program

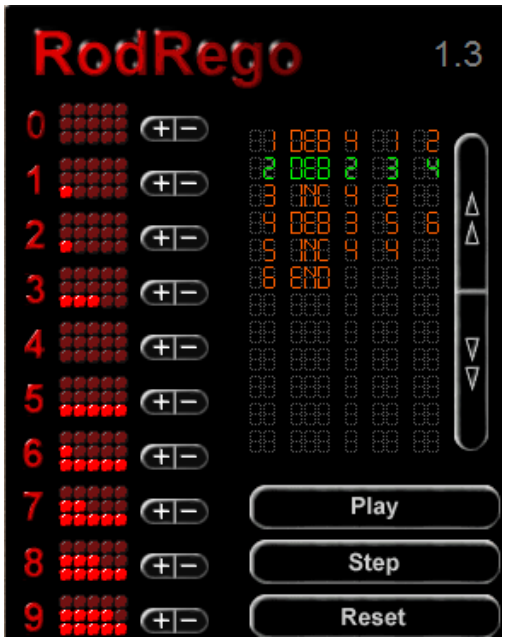
```
1 deb 4 1 2
2 deb 2 3 4
3 inc 4 2
4 deb 3 5 6
5 inc 4 4
6 end
```

This confusion is exacerbated by users having to enter programs in (1) without indication of what each column corresponds to. With no help, it is easy to mix up which numbers relate to instruction numbers and which to register numbers. It is also not clear that each instruction should have a number in the first column

Finally, the saved file format (.rgo) is specific to RodRego. Despite it containing a text based program, it cannot be opened in a text editor. This reduces flexibility; the files cannot be printed, and are not compatible with other programs, limiting sharing.

Running Programs

Figure 11 - Loaded, running program



When 'Play' is clicked, the program is loaded into (3) and, as program runs, the current instruction is highlighted in green. This feedback makes progress through the program clear to the user. The lights (2), representing data, flick on and off as the register is incremented or decremented. In this way, the system status is obvious to the user at all times.

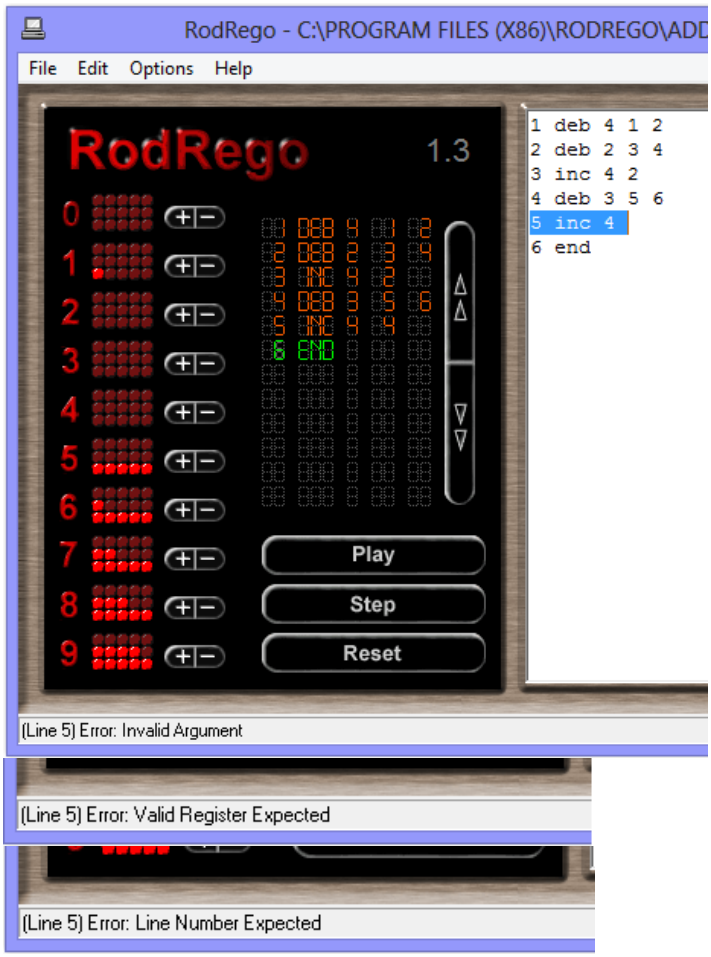
The user can choose to step through the program, or run it in its entirety; this eases debugging.

But the effects of running a program can be difficult to follow. There are ten register representations on the interface and new data is displayed with the same red light as data already in the register. Decremented registers are not highlighted to show data was removed. There is no clear connection between the program in (3) and the registers in (2); there are no column headings to explain whether a register or an instruction is impacted. Even though the running program itself is easy to follow, its impacts are not. Another limitation of the register representation, not necessarily as apparent to users, is that numbers over 15 are lost. No error message alerts users that a result is incorrect due to this.

There is no easy way to reset the registers, and the user must manually add or remove data with the plus/minus buttons (4), one click at a time. The reset button (5) resets the program back to the first instruction, and not the registers, which is frustrating; to run the same program multiple times, with the same outcome, the user must update the appropriate registers each time. This makes troubleshooting an arduous task.

If there are any errors in the program, error messages are displayed. Some of these are clear, pointing directly to the affected line and stating the issue; "(Line 5) Error: Valid Register Expected". However, in other cases, they are not as straightforward; "Invalid Argument" does not provide any specification for what a valid argument is. Examples of these error messages are noted in Figure 12.

Figure 12 – Error Messages



If an instruction is entered incorrectly, a blue highlight of the instruction in (1) shows where the error lies.

Some of the error messages are clear, but none have suggestions about how to fix the error.

There is some functionality which must be retained in an updated RodRego system; the ability to save and load programs, the clear step-by-step progress through a program, and the relatively minimalist interface.

But, there is much that needs to be changed. The design must be updated to conform to usability standards. Not all registers need to be displayed at once as this is unnecessarily distracting. A direct connection between the instruction being completed and the impact on the registers would much clarify the action of the simulator. Tutorial and help sections would improve user understanding without having to refer to the academic paper, and labelling what is expected for each argument of an instruction is essential. Manually typing in instructions is prone to errors; this could be improved by an interface similar to iURM, and clearer error messages that provide solutions. How difficult it is to undo actions in RodRego dissuades users from trouble shooting and trying different tasks, this must be changed. These updates would expand the intended audience to include novice users.

Requirements

The overall aim of the project was to take RodRego and implement it on a modern platform. A narrowing of focus was required to ensure a well-defined project with clear requirements. These were gathered through discussions with Dr. Simon Rogers, a review of background literature and current solutions, and evaluation of prototype designs with experienced users. A full list of the requirements is in Appendix (X), arranged as per the MoSCoW method, a tool to prioritise tasks. These are divided into priority groups, indicated by the assigned letter; Must (M) is absolutely critical to the success of the project, Should (S) are important, but a solution would still be viable without them, Could (C) are desirable, and do not have a major impact if left out and finally, Would (W) are requirements that future iterations of the solution may have, but they are not a part of the current process. (reference). This allows for the definition of scope and the correct assignment of time and effort in a project.

Apart from the fundamental requirement of modernisation, whilst maintaining programming functionality, RodRego is an educational tool and this shaped the avenues of research. This led to the decision of deployment to a mobile platform, and heavily influenced the interface. Many of the "should" requirements stem from the needs of naïve users, including in-depth tutorials, measures to reduce user error and parameterised error messages. These were sourced from design and usability heuristics for products with naïve users.

The evaluations that were carried out on current solutions also highlighted the need for this, as confusion and inconsistency negatively affected the learning outcomes of the applications. The lack of help, inability to undo, having to manually type programs, unintuitive interfaces, unclear error messages and lack of labelling were issues which had to be resolved, while keeping positive functionality such as clearly stepping through a program and saving/loading.

Finally the prototype design review cemented the tutorial requirement, with multiple evaluators mentioning it. The design presented was, in general, well received; this was probably due to it incorporating design and usability guidelines.

It was decided to take advantage of research indicating the effectiveness of gamification elements on the motivation and learning outcomes of users, and merge some of these elements into the application. Although not necessary for the success of the project, this was high priority. A register machine simulator is not innately compelling to most, and encouraging multiple visits through challenges, whilst providing positive reinforcement, was expected to make it more captivating.

Despite investigations pointing to storytelling as a powerful way to convey complex concepts, it was decided that applying a metaphor or story to the tool could distract, removing the visualisation even further from the theory of a register machine. Simulating a register machine, with symbols for instructions and memory locations and allowing a user to create programs with these symbols, is already an analogy in itself, a step away from writing assembly language code. This was compounded by the language that needed to be used in the

tutorial, as highly technical terms had to be avoided to prevent confusion, again distancing the tool from core concepts.

Keeping in mind users should progress, leading to different expectations of application functionality, some requirements to fulfil their changing needs were considered. Subroutines and the ability to comment specific instructions were the two main functions to be integrated. However, due to the possibility of time constraints and the overarching “educational tool” requirement, these were marked as desirable, rather than critical.

Saving and loading of files, which allows users to continue working on unfinished programs, added the additional advanced functionality of letting users manually create their own text programs. The requirements were drafted to ensure the files were created in a widely used format and defined structure, with the programs written in Dennett's RAP (seen in Figure X). This meant they could be edited outside of the application, expanding the user experience, and further deepening their understanding of register machines.

//TODO – USE CASE DIAGRAM

Design and Implementation

After the requirements were defined, classes were designed, along with a prototype of the interface. An evaluation was carried out on this prototype, with feedback integrated, before the interface was implemented. At the same time, the Java classes were developed. Although it was expected that the interface would not have major impacts on the Java code, some design decisions did have effects. Next, further evaluation was carried out with naïve and experienced users; this focused mainly on the ease of use of the interface. After the captured observations were updated, a small number of long term evaluations took place to assess the aspect of learning; a requirement of the tool is to convey register machine theory, and connect this with CPU functionality.

Programming was completed in Eclipse, with GitHub version control. An Xperia Ray, Samsung Galaxy s4 and Motorola Xoom tablet were used in development testing. Tools such as HierarchyViewer were used to assess Android Layouts.

What follows is a detailed overview of the design and implementation process, starting at prototype design and evaluation, moving onto the Android and Java implementation, and then on to the two user evaluations of the application. A final section discusses the unit and system testing which was completed.

Interface Design and Implementation

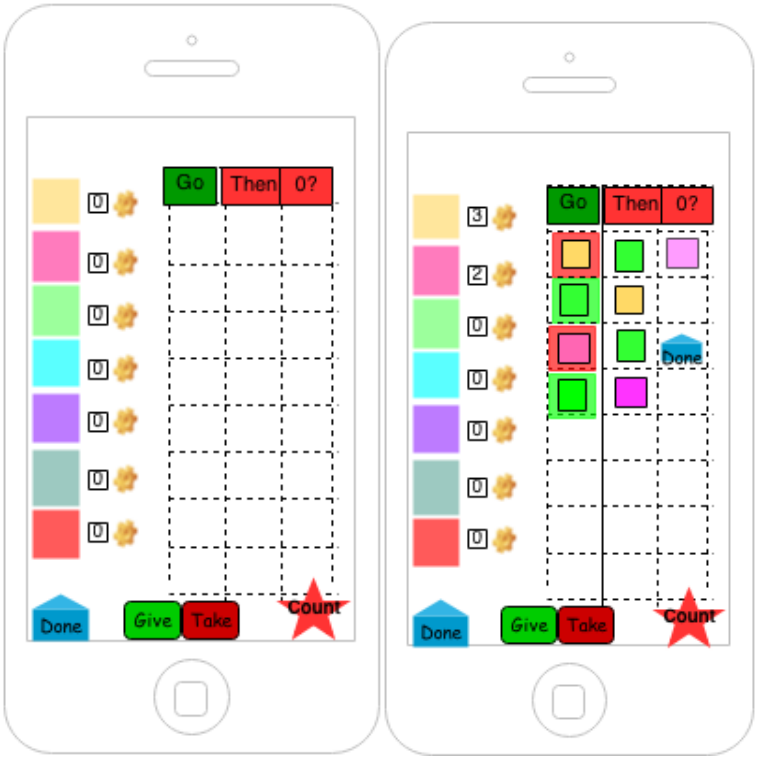
Due to the importance of the interface in the success of this project, much research in guidelines and evaluation was completed. Firstly, multiple interfaces were designed, and programs manually created using these. When it was felt that a strong contender had been developed, the prototype design was evaluated by volunteers with computing backgrounds. This helped assess how appropriate the design was, and if there were any flaws in usability. There were no major issues, and this interface was implemented. When completed, it was subjected, again, to rigorous user evaluation to determine if it was actually usable on a device. Feedback from these users changed various aspects, including some important visualisation elements, before a final, third evaluation with completely naive users, the target group for the application. The following sections detail the design and evaluation of the prototype, and implementation in Android.

Design of Prototype

The visualisation of the instructions and how they would be connected, allowing for many different kinds of programs, was a complicated decision. The first thought was to use Dennett's flow graphs, the user dragging and dropping increment/decrement instructions on screen and connecting these with loop and branches. However, it was felt that this would not sufficiently convey the sequential nature of a program to users without a computing background. Also, with three different connections required (next, loop and branch), all crossing over and under each other, the program on screen could become unintuitive and confusing; literal "spaghetti" code. The user should be assisted to keep the program as visually clear as possible. Setting instructions consecutively in a line or table reduced the connections required to two; loop and branch.

The first prototype is shown in Figure 13. This concept incorporated a storyline, with the user "Giving" and "Taking" cookies from coloured houses (registers). Each row was a step; the coloured box is the connection to the register, and the background colour indicates give or take. The program would complete the give/take and move to the step indicated in the "Then" column, or branch to the step indicated in the "0?" column, if the register was zero. The program in Figure 13 adds by moving the data from Register 1 and Register 2 into Register 3.

Figure 13 – First prototype design, Cookie Application



This integrated many of the design and usability guidelines, as well as some positive aspects noted in the evaluation of current solutions; there is minimal text, with only the required options on screen. The interface is visual interactive, and balanced, with reminders as to the actions of the buttons. By using a table, errors are reduced; no instruction could have two next steps.

However, prior to evaluation, issues were noted. If a register is affected by two steps, it is not clear to the user which of these two steps is being branched or looped to by other instructions. A semi-transparent overlay of connecting lines was a possible solution but, similar to flow charts, could lead to confusion. Also, the user would have to specify the next step in the "Then" column for all rows. This additional procedure could frustrate users, as it is not intuitive to have to state which step to go to if it is next in the table.

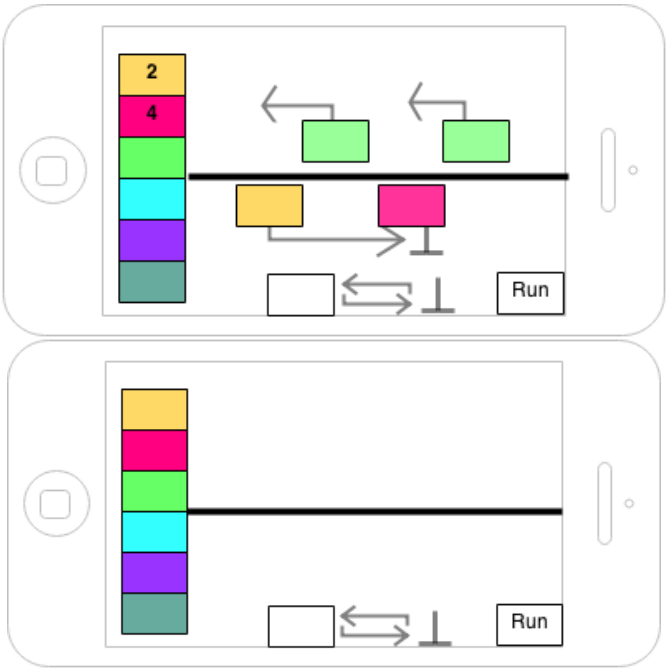
For the evaluation, a different prototype was designed. This is shown in Figure 14, and is similar to the final design of the application.

This did not have a story line, for reasons previously discussed, and builds programs along the line across the centre of the screen. The box icon places an increment or decrement instruction on the line, depending if it is above or below, and the arrows represent the looping or branching. An arrow above the line loops back, and an arrow below the line branches forward. The program shown is the same addition program as in Figure 13. Tapping the registers on the left adds number data; the operands for the program.

The icons are recognisable, without needing text, and the interface is minimal, but interactive. Only the required media is on screen, reducing distractions. User error is reduced by only being able to add instructions with the icons. Unlike the Cookie prototype, the instructions

within the program have an obvious successor; the next instruction on the line. The program itself is clearly differentiated from the registers and icons. Both of these make this program visually clearer. The user does not have to manually set the successor, as it is automatic; fewer steps mean fewer chances for error. Finally, by not using an overlying metaphor, the tutorial can be less obfuscated.

Figure 14 – Second prototype design, Line Application



Evaluation of Prototype

Six users evaluated the prototype design in Figure 14. Five of these were MSc. IT students who had completed Systems and Networks, a course which covers CPU functionality in depth, as well as assembly language programming, and one was an MSc. CS student, with a background in computing science. On a scale of 1-5, most considered their knowledge of CPU function and register machines to be a 3.

The aims of this evaluation were two-fold. First, feedback on the design and program building functionality would indicate if the interface was useable. Secondly, open ended questions about register machines and the proposed design could gather additional requirements.

The surveys are found in Appendix E.

Methodology

These users were given an overview of the project and register machine functionality, followed by an in depth description of the prototype. They were then asked to complete a questionnaire, and were encouraged to “think out loud”. They were prompted to comment on

various aspects of the design and functionality, rather than providing short answers on the questionnaire

Results and Feedback

No additional requirements were noted from questions probing the volunteers for good and bad aspects of register machine simulators they had used.

Good feedback was received overall; each evaluator agreed that the application conveyed the concept of a register machine, the steps to build programs were straight forward and that the complex concepts were simplified. One user liked the fact that the application "Uses simple concepts such as shape, colour and movement to represent more complex ones, making it easier to understand how a register machine works"

The layout was considered to be clear, colourful and simple. Multiple evaluators felt the application would be useful, with one specifically stating "Wish we'd had something like it in class". All thought the level of difficulty was appropriate for naïve users, and that the application would increase their knowledge.

Some feedback was received:

1) Intensive tutorial/help required.
2) Gaming aspect required – challenges, for example.
3) Character to help users through tutorial.
4) Colour arrows to clarify where they come from and point to.
5) Clear distinction between the arrows (branch/loop).
6) Motion of program unclear – animate the line from left to right/use chevrons.
7) Possibility to add additional registers.
8) Add breakpoints.
9) Sound cues.

Feedback on the tutorial and gaming aspects strengthened the requirement for these (1-3). The feedback for the arrows was not incorporated immediately, as it was felt that this might add additional complexity to the interface, and another tester believed the arrows were already too emphasised (4-5). It was decided that updates to the arrows would not take place until user testing of the developed application was completed.

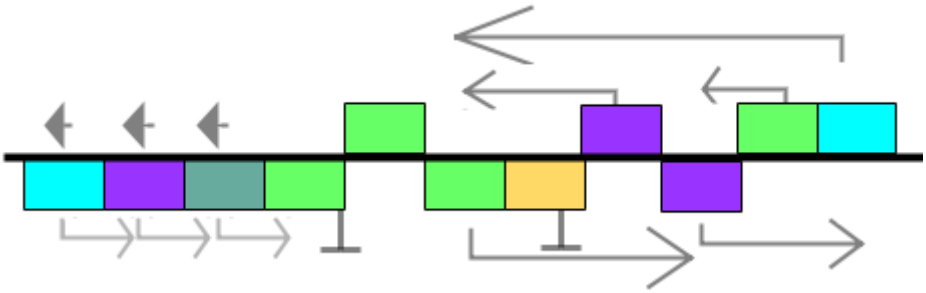
It was thought that animation of the central line would prioritise the sequential nature of the program visualisation too much, taking away from the major role that arrows play. Therefore, this feedback was not used, but it was noted that the tutorial should stress the left to right movement of the program (6).

Adding additional registers and breakpoints are functionalities which would appeal to advanced users (7-8). As this project's main aim is to develop an educational tool, within a limited timescale, these requirements were considered low priority.

Finally, as research showed that text, pictures and audio together can actually reduce learning, the suggestion of sound cues was not integrated (9). (reference)

The positive feedback and ability to create visually clear programs, even complex ones like division in Figure 15, indicated that this would be a reasonable design to implement. The decision was made to move forward with this design.

Figure 15– Division program, Line Application



This program ensures the divisor (Register 3, green) is not zero, and then decrements the divisor and dividend until the divisor is zero. It then restores the divisor using the information in Register 5 (purple), captures the full decrement in Register 4 (turquoise), and returns back to the start of the loop.

Implementation of Prototype

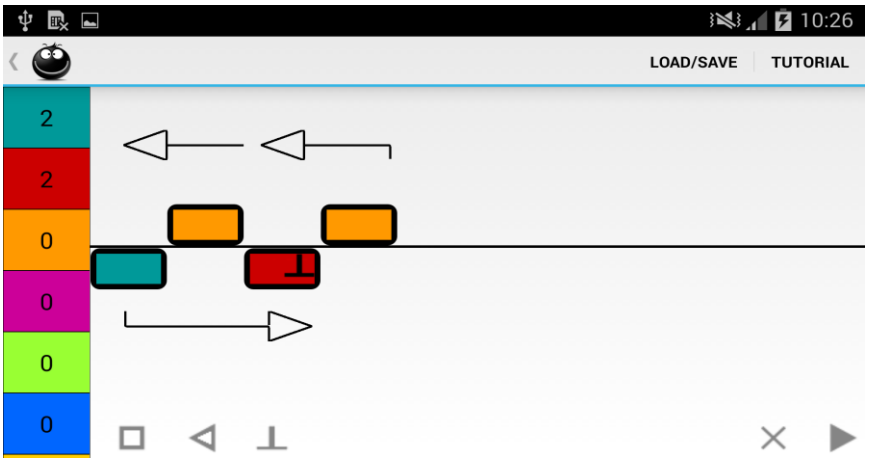
The user interface of an Android application is composed of both XML layouts, stored as static .xml files, and dynamically added elements. When the application is launched, `onCreate()` is called in the first Activity, the launcher. This method inflates, or renders, an XML layout on the screen.

The XML layout defines nested ViewGroups and Views. ViewGroups are containers, holding child Views. They can apply attributes to these children, and the children themselves can also have attributes. ViewGroups and Views can be assigned IDs. Anywhere in an Activity, a ViewGroup container can be called with this ID, and a new View added to it with its own layout parameters.

Android layouts occur in two passes; the first measures ViewGroups and Views, a top down traversal, where parent ViewGroups push measurements to child Views. The child View then sets its own size, as per its attributes, and the measurements it received. For example, a child View with the attribute of `android:layout_height="match_parent"` will size its height to the largest possible allowed size. The second pass actually places the ViewGroups and Views on screen. (Reference)

The developed user interface is in Figure 15 with an add program. This is prior to user evaluation of the implemented application.

Figure 15 – Developed Interface, prior to user evaluation



Overall Structure

The overall structure of the application interface is a `RelativeLayout`, containing two ViewGroups. To the left, there is a `ScrollView` for the registers. Each register is a `Button`, their colours and data populated dynamically in code. Across the top of the screen is the standard Android `ActionBar`. The rest of the screen is taken up with a `LinearLayout`, to allow a consistent 85/15 split between the command bar at the bottom of the screen and the central area for programming when it expands and requires scrolling. This is achieved using `android:layout_weight` attributes.

The command bar contains the "Create Instruction" icons of box, arrow and end, with clear/bin and run icons also, each a static `ImageButton`. These are invisible or visible, depending on the system status. This assists the user, as the arrow, end, clear/bin and run icons appear only when they are actually applicable. To reduce the elements on screen, the clear icon becomes the bin icon when an instruction is dragged.

ActionFrame and Instruction Placement

The application programming area, or `ActionFrame`, created some interesting programming issues. This area is a `RelativeLayout`; each instruction must be placed relative to other instructions, and the horizontal line, using relative layout attributes. Iteration through the list of instructions, in order of the steps of the program, places each instruction on screen separately. In order for a View to be given a relative layout attribute, the relative View must be on screen. Two major problems were dealt with when it came to the arrow instructions.

The initial box `ImageButton` (increment/decrement instruction) is always placed relative to the `ActionFrame`; left of this parent ViewGroup. It does not rely on other instructions for its placement. For the next box instruction, adding the layout parameter of `(RelativeLayout.RIGHT_OF, previousBox)` to the `ImageButton` is sufficient because the initial instruction is present before the `ImageButton` is placed on screen. An arrow must be placed relative to these boxes; spanning the correct amount, and relative to other arrows. However, the arrow may be pointing to a box instruction whose

ImageButton is not yet on screen or, similarly, it may overlap other arrows not yet added. This caused null pointer exceptions, and was the first issue to resolve.

Secondly, a custom View was originally used for the arrows. Custom Views must override onMeasure(), a method called by the parent ViewGroup during the first layout pass. This allows the custom View to set its measurements for actual layout in the second pass. The size of the arrow custom View depends on the box instruction visualisations it interacts with. But, the boxes were not yet on screen, as this is completed during the second pass and, so, the custom View returned a size of (0, 0). This made the arrows invisible. Performing multiple iterations to add Box instructions first and Arrow instructions second, for example, was not a solution as all child Views are traversed for all layout passes.

These issues were resolved in two ways. The first was to use a standard ImageButton for each arrow and include a Sizer on screen; an invisible ImageButton which matches the box ImageButton size. The Arrow.class checks the number of boxes it spans by iterating over the list of instructions. The DrawArrow.class uses this to create a Bitmap of the correct size, multiplying the number of boxes by the Sizer width, and draws an arrow onto this Bitmap using a Canvas. It is placed on the ImageButton which can, in turn, determine its size from the Bitmap. This ImageButton is then aligned relative to the arrow's preceding box ImageButton, which is always on screen prior, due to the sequential nature of adding instruction visuals to the screen. This resolved sizing issue and the horizontal layout, relative to the boxes. However, it gave rise to a vertical layout error; the loop arrows cut through increment instruction (above the line) if they were associated with a decrement instruction and spanned multiple boxes.

Aligning the arrows vertically was resolved, again, with the Sizer. An arrow instruction ImageButton is the height of the Sizer. Iteration over the full list of instructions determines if there are any increment and decrement instructions; MainActivity.class, as it is providing the instruction visualisations with layout parameters, can then set each arrow ImageButton with a default margin to avoid cutting through a box ImageButton.

Then, MainActivity.class compares the spanned instructions of arrows against each other. If there is any overlap, the margin for the ImageButton is increased. These were sufficient solutions, and the arrow instructions correctly lay out on screen, without overlap.

2D Scrolling

For longer programs, the onscreen visualisation becomes too wide for the screen, and, in some cases, too tall. Without an integrated 2D scrolling layout in Android, TwoDScrollView, a custom ViewGroup, was used. This wraps around ActionFrame, becoming its parent. Although extremely useful, TwoDScrollView extends FrameLayout, and not ScrollView; this is an issue when it comes to sizing children within the TwoDScrollView.

If a child of a ScrollView is set to match its parent, the ScrollView will not activate; the child View will constantly expand to the height of the ScrollView, rendering it useless. Data in the child View will be pushed off screen. To counteract this, ScrollView

can be set with an attribute, `android:fillViewPort`. This forces the child to fill the viewport, not the parent `ScrollView`, allowing the parent to activate scrolling. This same attribute can be used in any layout which extends `ScrollView`.

The horizontal line across the application programming screen, `theLine`, is centred within its parent, `ActionFrame`. In order for `ActionFrame` to centre this child View, it must match the parent height to determine a central point. However, as `TwoDScrollView` does not have the necessary `android:fillViewPort` attribute, which would allow for `ActionFrame` to expand correctly, it cannot actually ascertain this central point. All child Views of `ActionFrame` were impacted by this; they were relative to each other, but off screen, as `theLine` was placed, by default, at the top of the parent view.

A discussion with Dr. John Williamson of the Glasgow University School of Computing Science uncovered three options; use a different 2D scrolling layout, implement an entirely custom ViewGroup for the interface or execute the `ActionFrame` as an `AbsoluteLayout` instead of `RelativeLayout`, with exact coordinates for each child on screen.

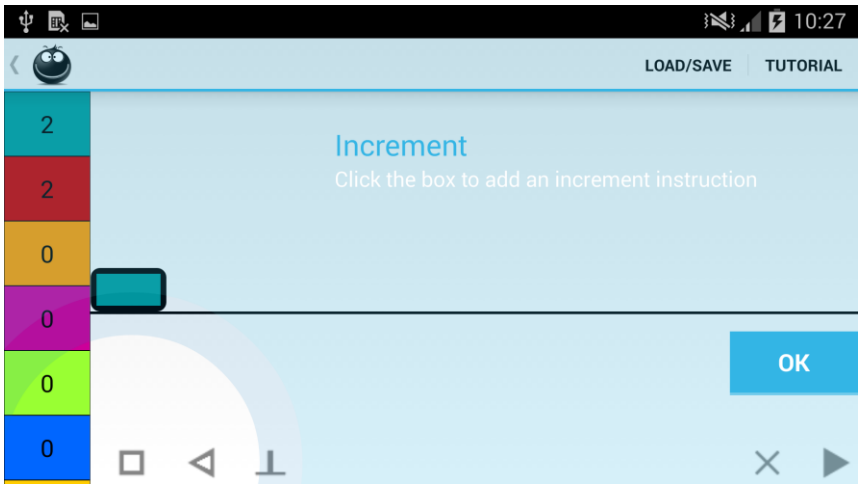
Investigation yielded no other 2D scrolling layouts and, due to time constraints, the design and implementation of a custom ViewGroup was not feasible. The suggestion of `AbsoluteLayout` was initially thought to not be good programming practice. However, it is a reasonable solution for this type of dynamic interface; the elements are placed relative to each other and the parent layout, to visualise a program. Their placement is not static, and relies solely on the other elements on screen. Therefore, as long as the coordinates are determined each time the program is drawn, as opposed to being hard coded, the interface remains accessible to different device screen sizes.

`AbsoluteLayout` is, unfortunately, deprecated but a similar resolution in the form of margins was achieved. The instructions are provided with layout attributes of above or below `theLine`, depending on their type. Then, using the `Sizer` width and height, the margins of the elements, relative to `theLine`, are set. The `ActionFrame` can then use these two attributes to accurately place the child View on screen. As previously mentioned, the margins for an arrow instruction `ImageButton` is also impacted by other elements on screen. Additionally, the `ActionFrame` was set to a fixed size. This resolved the issue of scrolling not activating; the `TwoDScrollView,` by default, must scroll to fit this fixed size.

Tutorial and Error Messages

Another aspect of the interface is the tutorial. This has a simple XML layout, consisting of an image, some text, and buttons. It integrates the library of ShowcaseView v5.0 (reference), a tool to easily highlight important parts of an application to the user. Figure 16 show this in action.

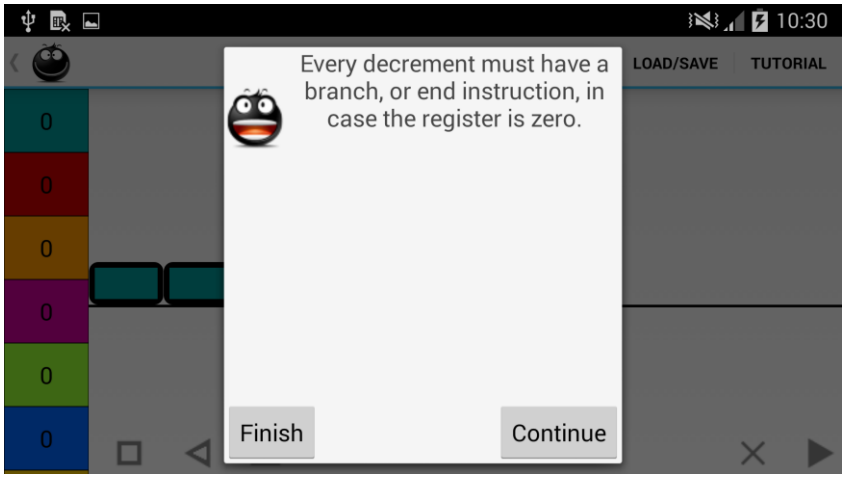
Figure 16 – ShowcaseView



ShowcaseView uses the Builder pattern to implement this screen. It is provided with the target for the focus circle, the title and text, and a click listener for the "Ok" button. In the case of this application, it was further configured to prevent users pressing outside the focus circle, and moving the "Ok" button position so it did not cover the clear/bin and run icons.

As the user moves through the tutorial, a particular dialog is presented, depending on the position in the tutorial. These contain an image of the character, "Reggie", and some text to guide the user. The tutorial then displays a ShowcaseView to invite the user to interact with the application, an important part of active learning. The tutorial texts explain a register machine, link this with CPU functionality, define the instruction set, and walk the user through the steps required to build a program. It also provides an explanation where something may be unintuitive, as in Figure 17; to a naïve user, the requirement for a branch/end may not make sense.

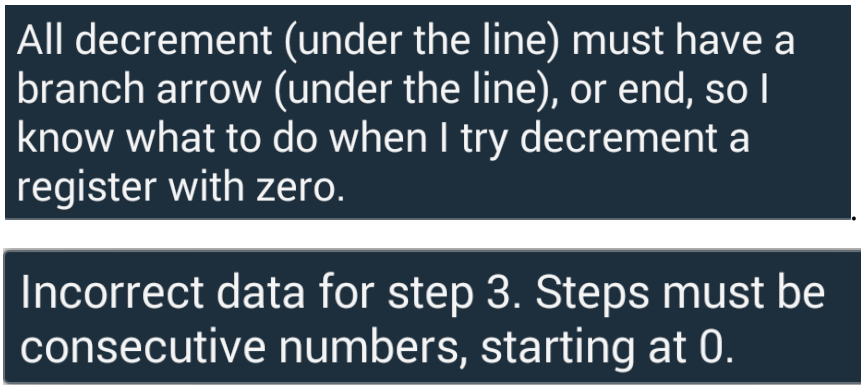
Figure 17 – Tutorial



This in depth tutorial consolidates the background research and requirements for the application, in the hope it provides an introduction to CPU functionality, and assists learners.

Another requirement was useful error messages with steps to resolve the error. These are throughout the application; file import, file export, building programs and running programs. Each is parameterised either by virtue of its position within the program, or dynamically from the code and helps the user to deal with the error. Two examples are seen in Figure 18. The first warns the user about having no branch or end for a decrement instruction, and the second occurs if a row on a program file import has a number which is not in order, as the step numbers must be consecutive. This is parameterised by the expected step number.

Figure 18 – Error message examples



As documented as a risk in the proposal for the project, the Android programming took longer than expected. This was due to the inexperience of the author but also, in part, due to the complexity of the visualisation of the program. Android layouts were not designed for constant, dynamic updates to a 2D scrolling screen, due to the two pass layout process and the lack of a 2D scrolling layout. This type of application is usually implemented using a `Canvas`, capturing the velocity and direction of a scroll and redrawing elements, or, as was suggested by Dr. John Williamson, a completely custom ViewGroup.

More research prior to programming would not necessarily have highlighted the planned approach as an issue, as `RelativeLayout` works well when wrapped within a one directional `ScrollView`, once the two pass layout process is understood, and `TwoDScrollView.class` is, at first glance a fully working solution.

Unfortunately, this led to the removal of gamification requirements from the project, as these were not interlinked with other requirements, and were not critical for success. Attempting to implement these would have had a negative impact on the critical success factors, including the indispensable user evaluation.

Java Design and Implementation

Prior to starting programming, class diagrams were developed. These evolved over the implementation process, due to unforeseen issues and updates to how the application would run. The final class diagram can be seen in Figure X.

From here, a description of the design pattern is detailed, followed by a general overview of the patterns and programming practices with retrospective critique of the code. Then, an overview of each class, and its interactions with other classes, including a discussion of challenges which were overcome in the programming process.

Design Pattern

The class structure follows the Model-View-Presenter (MVP) design, an iteration of Model-View-Controller (MVC) which is better suited to Android development. (reference) Central to both is the requirement to keep presentation separate from logic, allowing for easily maintained, modular interfaces, but there are some differences. A class diagram can be seen in Figure X.

The Model is the same in both, holding the structure of the system data and updated by user interaction; this application has `Box.class`, `Arrow.class` and `End.class`. These are generalised to an `Instruction` interface.

The View differs; an `Activity` is the "gateway" to an application, which would make them MVC controllers. However, classes which extend `Activity` must call `setContentView()`. This method is parameterised with an XML layout, and renders it on screen, tightly coupling an `Activity` and an XML layout. An `Activity` must also deal with events such as `onCreate()` and `onPause()`. This naturally leads to localisation of user event routing through the view. `MainActivity.class` is the central View, with others like `Tutorial.class`, `SplashScreen.class` and `OptionScreen.class` taking some specific responsibilities.

Finally, there is the Presenter, which is often referred to as the man in the middle – `Game.class` in this application. This retrieves data from the model, formats it, and provides it to the view. This decouples presentation from data, allowing for flexibility; a new GUI module can slot in, extending application use across multiple devices. It also has control over the effects of user interaction. `Game.class` also instantiates helper classes `ErrorCheck.class` and `SaveLoad.class` which interact directly with the Model classes, but do not update them.

//TODO – CLASS DIAGRAM

Programming Practices

Apart from the decoupling and separation of concerns implemented through the MVP pattern, best practices of object orientated programming were also followed, like using interfaces, getters/setters, and both Java and Android specific decoupling. A brief overview of these types of implementation decisions follows.

Generalising all instructions under one interface, `Instruction`, allows `Game.class` to quickly iterate over and pull information from the instructions without needing to interact with each instruction differently depending on its type. As much as possible, the application refers to `Instruction`, rather than the specific type. This allows for future extension; perhaps to include a Jump or Transfer instruction.

However, the application does use `instanceof` in many cases. This is not good practice as it does not take full advantage of Java polymorphism, and changes to algorithms may affect `Game.class` and the instruction classes. A better design would incorporate the Visitor pattern. A helper of `Game.class`, `Visitor.class` would contain all the algorithms which `instanceof` is currently being used for; for example, updating an instruction's position within the doubly linked list of all instructions. This `Visitor.class` would use method overloading, multiple `visit()` methods, each parameterised by one of the instruction types. When `Game.class` receives a user event to change the position, it would dispatch `Visitor.class` to its children, the instructions. Each instruction would have an `accept()` method that calls `visit()` within the visitor, passing itself as a parameter. Using this design pattern would pull all associated algorithms to one class, making it more easily maintainable. Each set of algorithms would then need a different `Visitor.class`.

Classes are decoupled; content coupling is reduced by keeping instance variables private, accessed only through getters/setters. Examples of this can be noted throughout all classes. Control coupling is eliminated by ensuring methods pull the data required to complete their actions, instead of being passed action flags; this can be seen in particular between the `Arrow.class` and `Box.class`, as the actions of `Arrow.doWork()` depends on what action the `Box.doWork()` was able to complete.

Some coupling is present in the code. Firstly, common data coupling occurs where constants, like the maximum number of registers allowed, is shared amongst multiple classes. This is minor. However, other coupling was required, and this impacted the MVP design that was being followed.

Any update to model data is completed through the presenter, `Game.class` but, in some cases, the data is accessed directly. In order to assign the correct layout parameters to an instruction `ImageButton`, the anonymous inner `CreateInstruction.class` needs to know the type of instruction that is being placed; arrow instructions have additional margins, for example. Also, during dragging and dropping actions, `MainActivity.class` captures the type of instruction, as the result of these actions depends on the instruction that is being interacted with. For example, when an arrow head is being dragged, it lengthens on screen. It is more efficient for `MainActivity.class` to

make these decisions, rather than passing the responsibility to `Game.class`, or keeping dynamically updated arrays of each type of instruction.

Best practice for Android is to keep all resources external to the application code. Resources like text, images and colours are stored in the `/res` directory. This allows for independent updates, and easy maintainability, as well as catering to different device sizes, and languages. Due to time constraints, this application does not currently have provisions for larger screens or other languages. However, because the `/res` directory was used, instead of integrating resources into code, future extensibility is merely a matter of adding the larger icons or translated text into the correct folder. The Android framework determines the device size or language, and pulls the applicable resource.

Finally, the instructions do not extend Android components, despite this adding additional code for connecting the onscreen component to the model data. This is for two reasons; firstly, extending the Android component couples the presentation of the data with the data structure. Secondly, the models are kept generalised and not specific to Android; they can be used in other contexts without any update. However, a balance between good programming practice and efficiency was struck for `Game.class`, `ErrorCheck.class` and `SaveLoad.class`. Android best practice is to use the `/res` directory, so parameterised error messages from Strings.xml are accessed with `getApplicationContext().getString(R.string.stringName)`. This is Android specific. `SaveLoad.class` also uses aFileDialog, an Android library, and `Game.class` has an anonymous inner class that extends `ASyncTask`, an Android specific thread framework. None of these would require large amounts of refactoring, if the interface was changed to, for example, Java swing, but they are not transferrable in their current form.

Class Overview

MainActivity.class

The main View, `MainActivity.class`, handles the majority of the layout and user event capture. It also contains an anonymous inner class, `CreateInstruction.class`, which iterates through the list of instructions, creates an appropriately parameterised `ImageButton`, and returns this back to `MainActivity.class` to add to screen. In general, it pulls information directly from `Game.class` and visualises this on screen; for example, the data contained in the registers. As discussed above, in some cases, it does capture information directly from the instructions, where it is efficient.

Due to the intensely interactive nature of the interface, `MainActivity.class` contains complex `onTouch()` and `onDrag()` methods. When a user presses on the screen, `onTouch()` captures the coordinates of this, and the start time. A short press updates the instruction; box instructions change their register, arrow instructions change their type. This is easily calculated by checking the start time against the end time of the user touch.

The `MotionEvent.ACTION_MOVE` dispatched when a user begins to drag an instruction passes responsibility to the `onDrag()` method. This is because using drag events allow for

finer control; `DragEvent.ACTION_DRAG_ENTERED` is dispatched when the dragged View enters the bounds of another View that has a drag listener attached. This is important for the arrow `ImageButton` extending along the instruction list as it is dragged, as well as dropping instructions into the bin. `onTouch()` does not have this finesse.

Depending on where a user presses on an arrow ImageButton, the action of the application changes; they can drag either the head, or tail of the arrow. This is calculated at the initial press.

The arrow head/tail calculation was fine-tuned multiple times over the course of the project. The raw (x, y) coordinates of a `MotionEvent`, in relation to the screen, are easily found. However, an instruction `ImageButton` does not have this attribute; its (x, y) coordinates are relative to its parent. This lead to complicated calculations, taking away the width of the `registerFrame` that holds the registers, checking the impact of the `ActionBar` at the top of screen and the command bar at the bottom. In addition to this, these calculations did not always work as intended; it is believed this was due to the scrolling of `ActionFrame`.

To ensure a stable calculation under all circumstances, the method of `View.getLocationInWindow()` is used. This returns the (x, y) coordinates of a View within the activity window; this is similar to `MotionEvent.getRawX()` and `MotionEvent.getRawY()`. From here, the calculations are straightforward; the left offset of the arrow `ImageButton` from the parent is determined. If the touch occurred at less than this offset, plus half with width of the `ImageButton`, and the arrow is a loop, the head of the arrow has been pressed. This new calculation led to fewer issues with dragging, but the main user frustration is still how awkward the arrows are to use. This could be resolved by a more complex algorithm, taking the velocity and direction of the drag operation and using this to determine if it is the head or tail; if moving left, drag the arrow head, for instance. Users with small device screens would also have fewer issues selecting the head or tail with this type of algorithm, particularly when the arrow spans just one box `ImageButton`.

It was found that, as many instructions were added to screen, the updates became extremely slow and the UI would freeze. To prevent this, `ASyncTask` is used. This is a helper class and performs background operations on a thread separate to the main UI thread. As a class must extend `ASyncTask`, and `MainActivity.class` already extends Activity, the anonymous inner class of `CreateInstruction.class` was developed. When an update to the onscreen program visualisation is required, `MainActivity.class` pulls the list of instructions from `Game.class`, passing the first one of these to `CreateInstruction.class` through an `execute()` command. This starts the `doInBackground()` method of `ASyncTask`.

`CreateInstruction.class` creates the `ImageButton` and sets it with all the layout parameters; how these are determined and the `DrawArrow.class` were discussed in the Interface Implementation section **(SECTION HERE).** After this has been completed, `ASyncTask` automatically calls `onPostExecute()`. This overridden method gets the next instruction from the list of instructions, creates a new instance of

`CreateInstruction.class`, and passes it the next instruction. This continues until all instructions in the list have been dealt with.

As `ASyncTask` is running on a background thread, it cannot add the `ImageButton` directly to the screen. This is only possible on the UI thread, hence `onPostExecute()` calls the `MainActivity.addToScreen()` method, which uses a `runOnUiThread()` methd t add the `ImageButton` to the screen. A similar issue gave rise to `MainActivity.updateInstructionDisplay()`, used by `Game.class` as the program is being run; each instruction is highlighted on screen, along with any impacted registers, and these changes must also run on the UI thread.

Game.class

The main presenter of the application, `Game.class`, holds most of the logic for the register machine. All instructions are added, edited and removed through `Game.class`, and the running of programs is the responsibility of its anonymous inner class, `RunGame.class()`.The chosen data structure for a program is a doubly linked list, each instruction containing a reference to its predecessor, and successor. This allows for efficient updating of the instruction list, as changing these references is straightforward; this efficiency of this data structure is required due to the extremely dynamic nature of the application. Also, `Game.class` does not need to hold the entirety of this list, just references to the first and last instructions; this is essential in memory constrained environments like mobile devices. (reference, android memory developers page)

As a general rule, when an instruction is added, it becomes the successor of the last instruction, an instance variable within `Game.class` that is updated dynamically when new instructions are added. Arrows must be provided with a “GoTo” instruction to ensure their correct placement on screen; this is the last box. Specific instructions can also be deleted from the list; allowing action undo is a crucial requirement for naïve users

Much thought and experimentation went into how instructions would be added and changed; there are certain standards that were clear from Dennett’s language, and others that had to be developed by the author to ensure a consistent user experience when running the game.

From reading Dennett’s RAP language, it is clear that each INC has exactly one associated register and exactly one “GoTo” step. Translating Dennett’s INC to an increment instruction means that the “GoTo” may be the next box in line, or it may be an arrow, pointing to an alternative “GoTo” step, depending on the successor of the increment.

DEB is more complex, as its “Branch” step is only followed under certain circumstances; this is why the standard was developed of a branch arrow always being a predecessor of a loop arrow, if they are coming from the same box. When the program is running, the `doWork()` method of box instructions sets the next instruction to be completed as their successor. The standard ensures that the branch arrow `doWork()` method will be called prior to a loop `doWork()` method; this method checks if the branch should be followed, or if it should be skipped.

This standard impacts much of the moving and changing instruction methods; if a branch arrow tail is dropped into a new box, `Game.class` must check what the current successor is of the box, and update links appropriately, if it is a loop arrow.

An anonymous inner class is used when running the program to give the user time to see impacted instructions and registers, highlighted on screen. This is completed by introducing a delay between calling `doWork()` on instructions. As is good practice, this delay occurs on a background thread, instead of freezing the UI. `RunGame.class()` extends `ASyncTask` to achieve this, and this allows the user to stop the program running at any time. A simple, future extension, easily implemented in this code design, is to allow the user to select whether they want to run the program fast or slow. This would allow for quicker troubleshooting. Currently, the delay length is a constant of `Game.class`. A second "Fast Run" button could be placed on screen, and selecting this would reduce the delay length.

Many checks are completed by `Game.class` as instructions are being changed on screen; branch arrows cannot branch to the same box instruction, as one example. Also, `Game.class` uses `ErrorCheck.class` before allowing a program to run, or a file to save; this iterates over the instruction list and performs sanity checks as well as standards checks; for example, every decrement instruction must have an associated branch or end, and two arrows of the same type coming from the same box is pointless, as one will never be reached. These error messages and checks are essential to reduce user error, an important requirement for the application.

Instructions

The models implementing the `<<Instruction>>` interface, `Box`, `Arrow` and `End`, together represent a program, their `doWork()` methods completing the updates to registers, and defining the structure and path of a running program.

Each is simple, holding a reference to its predecessor and successor and a unique ID which links it to the on screen `ImageButton` representing it.

The `Arrow.doWork()` method, as previously mentioned, is affected if a preceding decrement instruction has been completed or not. Pulling information about this allows the `Arrow` to set the next instruction to be completed as its successor, or its `toInstruction`; the `Box` instruction it points to. `End.doWork()` also completes a similar check, as an `End` instruction placed on a decrement instruction will only be followed if the associated register could not be decremented.

SaveLoad.class

`SaveLoad.class` is a helper to `Game.class`, creating .txt file programs from the instruction list, updating `Game.class` with new programs from .txt files and dealing with user input to capture saving and loading of files. Although this combines some of the presenter responsibilities with view responsibilities, keeping the code for these tasks combined in one class allows for easier maintenance.

`SaveLoad.class` uses aFileDialog, a file choosing dialog library for Android, which has been customised to ensure that users can only load .txt files. When a file is selected, it is run through multiple steps to ensure the format is correct. If, at any step, the data is malformed, the `Game.class` is not updated with new instructions. There are many error checks, comparing the format of the file to Dennett's RAP language; an example with column labels can be seen in Figure 1. Each row must be numbered consecutively, with less than 5 elements and an instruction (DEB, INC or END). The register, "GoTo" step and "Branch" step must be correct numbers.

As each error check is completed, another step on the way to creating a doubly linked list of instructions is also completed. An example of a file input can be seen in Figure X. First, an empty instruction array is created, the size of the number of rows; eight in the case of the program in Figure X. New decrement, increment and end instructions are created, using `Game.class`, to prevent duplication of logic, and placed at the index indicated by the step number.

Figure X – An example of file input, and output

```
0,DEB,1,0,1,
1,DEB,2,1,2,
2,DEB,0,3,5,
3,INC,1,4,
4,INC,2,2,
5,DEB,1,7,6,
6,END
7,INC,0,5,
```

These instructions are set as successors and predecessors of each other; if this program were visualised on screen at this stage, it would have no arrows.

The `SaveLoad.addArrow()` method updates the instruction list with the appropriate arrows; the "GoTo" and "Branch" steps point to the index of the correct `Box` or `End` instruction. If these are not the next step in line, an `Arrow` instruction must be involved; it is created, and the predecessors and successors updated. An `Arrow` cannot point to an `End` instruction, however, so there are some additional checks to ensure the `Arrow` is set with a `toInstruction` of a `Box` instruction.

Creating a .txt file is slightly more complicated. An array of `String` objects and an array of `Box` and `End` IDs are created; arrows do not have a step, as they are represented in RAP language by the "GoTo" and "Branch" steps.

The index of the `String` matches the instruction step number, and this index holds the ID of the associated `Box` or `End` instruction. Using `StringBuilder`, the step number, type of instruction and register are written to the String and stored in the array. If the successor is not an arrow, the "GoTo" step is merely the next step number, and this is also appended.

Iteration over the list of instructions again in `SaveLoad.arrowHelper()` pulls out the `Arrow` instructions. These contain a reference to their predecessor, and the instruction they point at; the ID of these is checked against the array of `Box` and `End` IDs. When found, the predecessor index equals the step number, and the index of the instruction being pointed at is

the “GoTo” or “Branch” step. The appropriate String can be accessed, and the “GoTo” and/or “Branch” step appended.

This class could be more efficient; firstly, a regular expression to match the expected file input for each row, instead of verifying each section separately, would reduce the amount of code. Secondly, iterating over the instruction list multiple times to create the .txt file is not ideal. The alternative is to write the instruction list to an array, and sort this; `Box` and `End` instructions would not change position, but all `Arrow` instructions would move to the end of the array. The instructions would need to implement the `<<Comparable>>` interface.

In this way, the `Box` and `End` instructions would not change step number, or position within the program, and the `Arrow` instructions would still point at the correct instruction. This array would only need to be iterated over once, the `Box` and `End` instructions being created first, and the checks mentioned above then being completed for the Arrow instructions, to append the correct details to the `String` at the applicable index.

Tutorial.class

A separate View of Tutorial.class handles all aspects of the tutorial, displaying text to the user using `Dialog.class` and highlighting features with ShowcaseView. It is completely decoupled from instructions, accessing information and adding instructions solely though `Game.class`. It does, however, interact with `SaveLoad.class`. This is because of the file loading ability of `SaveLoad.class`, allowing for any file to be populated at the end of the tutorial, instead of hardcoding the program on screen.

`Tutorial.class` uses ShowcaseView, an Android library that highlights a target on screen, prompting users to interact with the application. This library does have its limitations. As previously mentioned, it does not adapt to scroll text that is too large for the screen, and the size of the focus circle cannot be changed. It was not designed to be displayed in series, and this causes some issues with garbage collection. If a user clicks through the tutorial quickly on screen, ShowcaseView is not disposed of in time, causing an out of memory error. Forcing Android to perform garbage collection did not seem to resolve the issue. There may be other aspects to this error which further testing would reveal.

The user clicks on the dialog or ShowcaseView buttons to continue, this is captured, and a pointer incremented. Depending on the pointer, a `switch` statement shows the correct dialog or ShowcaseView. This caused some issues. The ShowcaseView must be provided with a target. At certain points in the tutorial, the target is a box or arrow `ImageButton`. The user is invited to create these, but if they don’t, ShowcaseView is given a null target.

`Tutorial.class` has to iterate through the instruction list at these points, check if the instruction has been added and, if not, call `Game.newInstruction()` to create it. Due to the two layout passes required to place the new instruction on screen, the creation of the new instruction has to be timed to allow these two passes to complete before the ShowcaseView can be provided with the target. In some cases, the new instruction is added as the previous dialog is shown, so as to not confuse the user.

A required extension for `Tutorial.class` is a "Back" button. This would decrement the pointer as a consequence of the user click event. The `switch` statement would then show the previous screen. `Tutorial.class` would also need to remove instructions added just previous, to prevent a large build up on screen, distracting from the target. This would require the storage of instruction IDs as they are added, and calling `Game.removeInstruction()` on the applicable ones; all boxes or all arrows, for example.

SplashScreen.class and OptionScreen.class

`SplashScreen.class` and `OptionScreen.class` are two simple Java classes; the first displays a three second picture of Reggie, branding the application. The second provides the user with a "Start Game" option. This can be easily extended in future iterations of the application to contain a "Load Game" option; displaying the aFileDialog which allows the user to navigate to a saved file, error checking this file and updating the `Game.class` with the correct instruction set are all responsibilities of `SaveLoad.class`. Code to instantiate the `MainActivity.class` and pass `SaveLoad.class` the associated `Game.class` to update with the new instructions would be trivial.

Util.class

To refer to specific Android Views in the Java code to, for example, add event listeners, the View must have an ID. This can be static, a `String` ID provided in the XML layout, which is converted to a 32bit ID by the Android framework, or it can be dynamically assigned in Java when the View is created; this assignment must be manually called. For this application, all elements on screen must have an ID, so user interaction with a specific element can be routed correctly.

In later Android OS versions, 4.2 and after, there is a method, View.generateViewId(), that generates an ID that does not conflict with the static IDs. For versions before this, however, another solution is required.

Within `Util.class`, the static method `generateViewId()` interacts with a thread-safe `AtomicInteger` to generate an ID. These IDs are clamped, as the Android generated IDs have a non-zero in the most significant bit, and will unique within a currently running version of the application. (reference)

Another way to complete this is have a static list of possible IDs as `String` resources, and assign these as the elements are dynamically generated; however, if the user added more resources than IDs, the application would crash.

`Util.class` is called by `Game.class`, when it creates new instructions. `MainActivity.class` takes this instruction ID and sets it as the instruction `ImageButton` ID, linking the visualisation with the model.

Evaluation of Application

Throughout the development process, there was regular evaluation of the application. The interface and logic were tested with both black-box and white-box testing by the author, and implementation issues and resolutions were discussed with the project supervisor on a weekly basis. As previously discussed, the prototype design for the interface was also assessed by evaluators.

After the development of the application, two types of user evaluation were completed. The first, short-term evaluation determined the functionality of the interface and application usability, and the second, long-term evaluation ascertained if users do learn from the application. A detailed description of the methodology, results and subsequent updates to the application follows.

Short-Term Evaluation

The first, short-term, evaluation was similar to the prototype evaluation. Users, both experienced and naïve, were asked complete the tutorial, interact with the application and fill in a questionnaire. The users were also watched, their interactions with the application noted.

The aims of this evaluation were to:

1) Test the interface design for clarity
2) Test the interface design for usability
3) Test if programs on screen are readable
4) Capture bugs not noticed during implementation
5) Test the tutorial's ability to convey the concept of the application, and how to interact with it.

Nine evaluators took part in the short term evaluation, using a Samsung Galaxy s4 (5", 1920px x 1080px display). The average time using the application was 20 minutes. This included completing the tutorial.

Their responses to the question, "What is your level of knowledge of how a CPU functions" are detailed in Table 2.

Table 2 – Responses to "What is your level of knowledge of how a CPU functions"

Scale	Users
1 – None	0
2 – Slight	3
3 – Some	5
4 – A lot	1

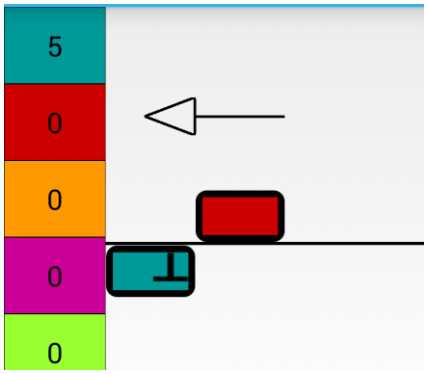
5 – Full	0

When questioned on how a CPU completes an "ADD" function, most were able to give a brief overview of the CPU taking the data from two registers, adding them, and placing the result in a third register. Some went into more depth, mentioning the use of adders, logic gates, binary and bits. One user in was unable to explain, simply writing "No idea".

Some users had never used a register machine simulator. Those that had mentioned inadequate or absent tutorials and the lack of a visual interface as frustrations, but did feel it was useful for learning assembly language programming.

After the baseline questions, the users completed the tutorial. At the end of the tutorial, a move program is displayed on screen, as shown in Figure 19.

Figure 19 – Move program



This program decrements the first register (teal), which originally has 5, and increments the second register (red), until the application attempts to decrement the first register and it contains zero. The program then stops.

The users were asked what actions this program took, and their responses (paraphrased) are shown in Table 3. Some of the users responded verbally, and others wrote their response on the questionnaire.

Table 3 – User responses for action of program

Response	Users
Correct – Decrement of first register, increment of second register, until the first register contains zero. Then the program stops.	3
Decrements the 5 to 0, and ends	1
Decrements and stops immediately	2
Decrements first register, increments second register, decrements first register and ends	1
Adds 5 to itself	1
No answer	1

Despite the majority of users not fully understanding the program, most mentioned decrementing. Some, especially the users who noted that the data in Register 1 was impacted would, perhaps, have determined the solution if given more time. All were able to follow the program when it was actually run.

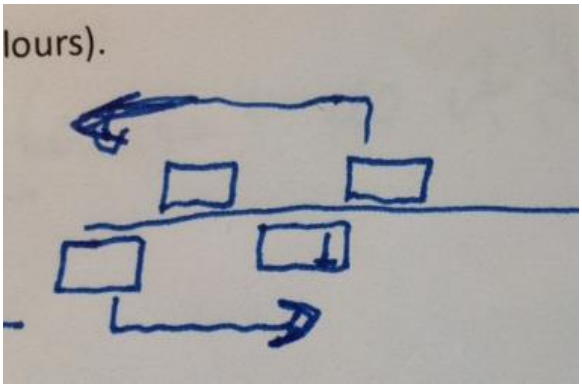
Interestingly, the user who specified “No idea” when queried on CPU functionality, correctly answered this question. This was a particularly positive outcome, as the application is aimed at completely novice users.

Overall, this suggested the visualisation was appropriate with a readable program, which conveyed the concept of the program itself, decrementing, incrementing and looping.

Next, the users were invited to spend some time playing with the application. It was suggested they attempt to build an “Add” program, as this was another question to be completed. They could also draw this, if they felt more comfortable.

Despite the short amount of time, two were able to create an “Add” program, with two others coming very close, needing only one additional instruction. An example is displayed in Figure 20. Again, this was a good outcome; the users understood the concept of adding using the limited instruction set, and were able to create this program using the application.

Figure 20 – User created “Add” program, with Reggie instructions; requires one additional arrow.



This program would work if the second register being decremented is larger than the first, as the increment of the third register would always be skipped when the first register reached zero.

If the first register is larger, the program would stop prematurely when the second register reached zero, never looping back to continue the loop of decrementing the first register and incrementing the third.

Finally, the users completed the questionnaire, which looked for feedback on all aspects of the application.

After using the application, most users were able to explain that a register machine is a simple CPU, with some describing the process in more depth. The user who originally did not know how a CPU worked responded with “Stores information and becomes the ingredients for functions [sic]”, a definite improvement in their knowledge.

The feedback on the interface was excellent; users mentioned that it was simple, sleek, attractive, colourful, and elements were clearly represented. The character of Reggie was well received, with multiple users listing him as a something they liked about the application.

It was mentioned that figuring out the program at the end of the tutorial, and build an addition program was fun and challenging. Another user compared the application to Sudoku. This demonstrates the importance of the gamification elements which, unfortunately, could not be fully implemented.

The tutorial was considered to be very clear by all. Some felt that the explanation of arrows and branching needed more description, and more than one evaluator requested examples of

programs in the tutorial. These improvements were implemented, but not to the exhaustive level that was suggested of walking through creating a program. There were two main reasons for this; first, some users felt the tutorial was already long, with one stating “I forgot how to do things” due to its length. Second, it is important that the tool challenge users, rather than provide all answers. This is essential in active learning.

In general, users did feel that they needed more time to play with the application to get full benefit, especially as instructions such as branch and loop were complicated concepts. A suggestion that came up numerous times was to have a “Help Mode”, where users would be provided with extra assistance when adding or dragging an instruction; for example, if the box icon was selected, a message would remind the user that they can drag the instruction above and below the line which means increment or decrement, and tap the instruction to change the associated register. This would have a short animation demonstrating this.

This concept was too time-consuming to implement in this iteration of the application. However, it is an interesting and useful suggestion, allowing the user to tailor their experience to their own knowledge. Due to the design of the application code, it would be a relatively simple integration; `MainActivity.class` would instantiate a `Help.class`, holding the animations and pulling the correct String messages from the Android resource folder. `SharedPreferences`, a framework which stores persistent application data, would contain a `boolean` value indicating if the user had the help mode switched on. `MainActivity.class` would capture click events and route them to the `Help.class` to display the media, if this `boolean` were true.

The arrows were the main source of confusion and frustration. In particular, where the arrows were pointing to was not clear and, as previously mentioned, the concept of branch/loop was considered complex. This was not helped by the fact the both branching and looping were represented by the same kind of arrow on screen, despite having two separate actions. It was felt that arrows pointed back, or forward, along the program and not to the box instruction. Interestingly, this was feedback also received during the prototype evaluation; with this further confirmation, it was evident that an update had to be made. It was also remarked that the arrows were “fiddly” to use, due to their small size. This caused issues with dragging and dropping.

Further, smaller errors were noted by the users and author. The “Create instruction” icons were not clear to all users, some attempting to drag the register representations to the centre of the screen to create a box. The ShowcaseView did not strongly focus the user's attention to the correct aspect of the screen. This was due to the almost transparent background, which was not different enough to the focus circle, and also, in part, due to the inability to change the focus circle size. When targets are small on screen, many are encircled by the focus circle, distracting from the real target.

An instruction remaining static while being dragged and updating its position only when dropped was confusing; the users felt they weren't actually dragging the instruction. Pressing the “Clear” button did not clear the register data, which was unintuitive, as it should provide a clean slate.

Finally, some errors were noted in the application, due to users attempting programs which the author had not considered. These were simple code fixes.

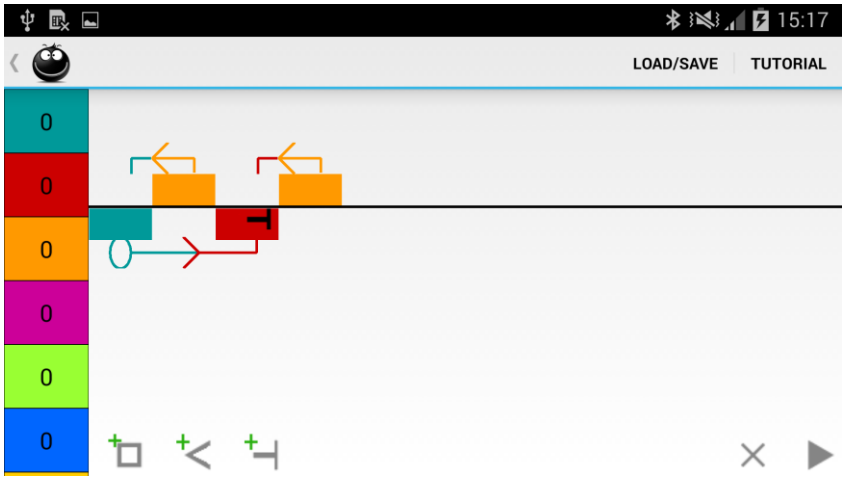
Taking all the feedback into account, the updates completed after user evaluation were:

1. Arrows heads/tails coloured as per the box they come from, and point to.
2. Arrow structure changed, improving clarity of from/to.
3. Loop arrow differentiated from Branch arrow.
4. Arrow size increased.
5. Arrow drag operation updated.
6. Green (+) added to the "Create Instruction" icons
7. Dragging Instructions causes them to disappear
8. More images of simple programs added to the tutorial.
9. Background of ShowcaseView darkened
10. Clear removes all register data
11. Minor text updates, code error updates.

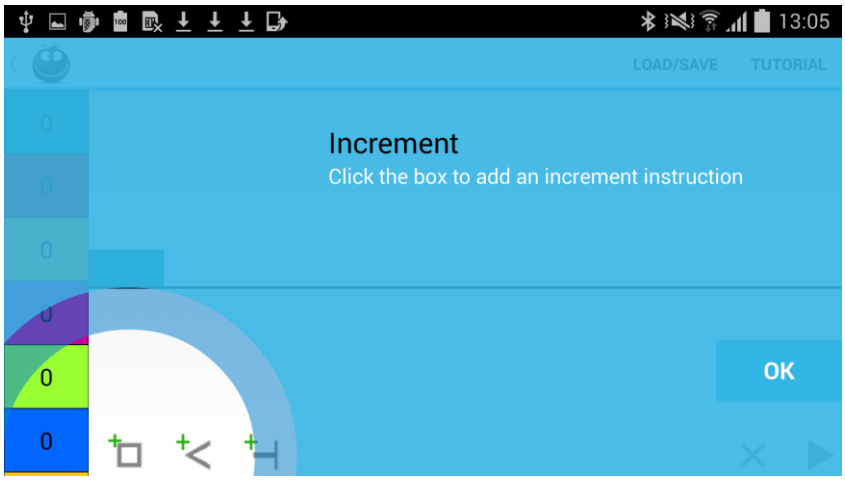
These updates can be seen in Figure 21. The updates to the Arrow drag operation are discussed in the Java implementation section (INSERT SECTION HERE)

Figure 21 – Updates to interface, after user evaluation

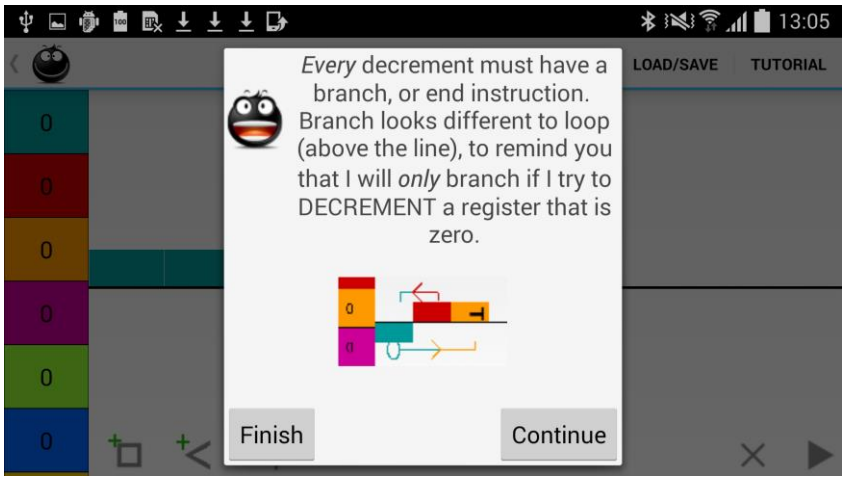
Updated interface, showing an Add program (Updates 1, 2, 3, 4, 6)



Updated ShowcaseView (Update 8)



Updated Tutorial, with pictures. (Update 6)



Long Term Evaluation

Due to the short amount of time the short-term evaluators were able to spend testing the application, it was felt that the learning outcomes would not be fully assessed. As an important requirement for this project is to deliver an educational tool, the second, long-term, evaluation was completed.

In this, users were provided with the application to download to their own devices. They were asked to use the application over the space of a week, and were provided with some suggestions for programs they could develop. This group also completed a questionnaire. The aims of this evaluation were to assess the learning outcomes after repeated uses of the application, as well as the four short-term aims.

Two evaluators took part in the long term evaluation, using their own devices, a HTC One S (4.3”, 960px x 540px display) and a Samsung Galaxy mini (4”, 800px x 400px display). Neither had any knowledge of register machines, nor had used a register machine simulator.

One user did state they had some CPU knowledge, and demonstrated this by giving a brief overview of how a CPU adds; “The contents of 2 memory cells are added by sequentially increasing one location by 1 and decreasing the 2nd location by 1 in a loop until the second cell reaches zero”. Due to the feedback regarding gamification in the short-term evaluation, these users were provided a list of possible programs they could try, ranging from easy (addition) to complex (subtract, including a check for zero). This was to help motivate them, in lieu of the gaming elements that were to be implemented.

They interacted with the application an average of 5 times, for 2.5 hours overall. Both used the app on lunch break and on their commute, confirming that the application does not necessarily need a formalised, structured interaction, and lends itself well to being played with for short amounts of time.

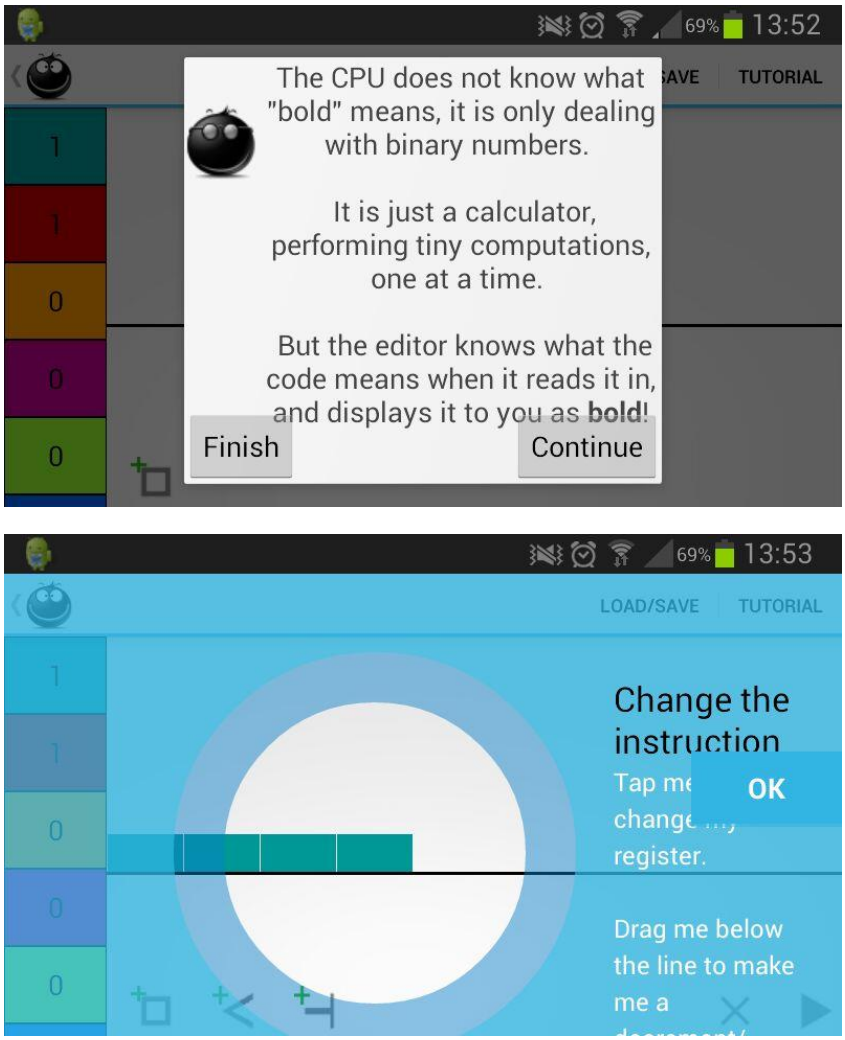
The feedback on the tutorial was similar to the short-term users, in that it was useful, clear, concise and straightforward. Again, the example program at the end of the tutorial was liked, and some further examples of complicated programs were requested. This would allow the evaluator to “reverse engineer” programs.

Both users found the error messages useful, a helpful reminder to create working programs. There was a suggestion of highlighting where the issue was on screen with the error message. This aligns with Neilson's usability heuristics (1995). The design of the code would allow for this integration relatively simply. `ErrorCheck.class` runs through the list of instructions, picking out errors. As this list of instructions is the same used to visualise the program on screen, a short method within `MainActivity.class` could be passed the ID of the incorrect instruction by `ErrorCheck.class`. Using this, the View on screen can be found, and highlighted, while the error message is being displayed.

The interface received excellent remarks but, again, the arrow representations were frustrating. Something which had not previously come up was the fact that, if the arrow was not pressed exactly, the screen would drag instead of the arrow. This is caused by the hard coded size of the `ActionFrame`, required because of the limitations of `TwoDScrollView`, and would be resolved by creating a custom View, or using `Canvas`, as previously discussed.

On the smaller screen, there were issues with `ShowcaseView`, shown in Figure 22. `ShowcaseView` does not adapt the size of the text or scroll the text when it is larger than the screen. This is a restriction of the current version of `ShowcaseView`, and caused some of the text to become unreadable. The same issue was noted with the `Dialog` text, also shown in Figure 22. To allow for a `Dialog` to scroll, the entire parent layout of `ActionFrame` must be wrapped with a `ScrollView`. A resolution to combine this wrapper `ScrollView` with `TwoDScrollView` without affecting the interface was not found.

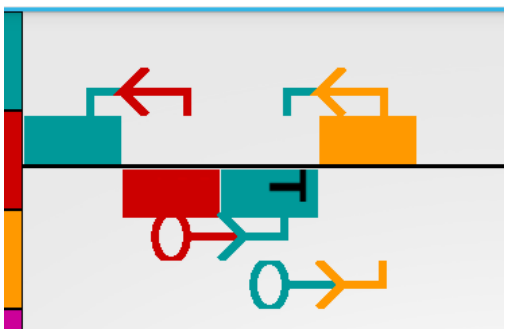
Figure 22 – Layout issues on smaller screen



The evaluators were asked to provide screenshots or .txt files of any programs that they wrote. The first evaluator provided their program screenshots in an email, and this is also found in Appendix X.

They were able to complete programs but not with the outcome they expected. A common issue noted with their programs was an increment as the first step in a loop. This means the loop completes one circuit too much, and the user had to compensate by decrementing the register when the loop was exited. An example of their program is seen in Figure 23

Figure 23 – User created program



This program is attempting to move the data from red to teal, and then from teal to orange.

This is almost fully functional; the first loop completes one circuit to much but, interestingly, the second loop is correct. The evaluator stated this program runs to infinity (email, 18Aug). This is due to the additional branch from teal to

orange which, due to the design of the program, is always reached prior to the end instruction.

The user was unable to troubleshoot the two issues with the program in Figure 23, and this was important feedback.

//TODO – Second user to send on their programs.

It was felt that the completion of one loop too many was not an issue with application functionality or clarity, but in the logic that the user approached the problem with. This type of "move" loop is shown at the end of the tutorial, where a user can watch its progress. Further troubleshooting or repeating the tutorial may have helped the user resolve this issue, and the "Help mode" suggested in the short-term evaluation could also provide additional assistance for simple programs. This user also stated they attempted to build a multiply program but, without an understanding of the basic programs, they were stymied. They suggested that a second, more advanced tutorial be provided for when all attempts at a program have failed; a similar suggestion to the "Help mode".

The second error, with the additional branch, highlighted that further error checks were required; it is not clear that an arrow will be reached by the program before an end instruction, meaning the end instruction will be skipped.

Finally, the learning outcomes were assessed. Both users did feel that their knowledge of CPUs had increased; it was felt that Reggie made the action of a CPU understandable at a basic level, a positive outcome. To assess if their knowledge had increased, the users were examined.

The users were asked to describe a register machine. Their responses were:

"Like a simple virtual computer, it can store numbers in different registers and can be programmed to add or subtract or hold data in those registers (but never goes into the negative integers)."

"A basic CPU, that stores numerical data in cells, which can be moved to other cells under add/subtract instructions, instructions which can be further combined in ways to perform more complicated tasks"

Both of these indicate an understanding of program creation by combining instructions, and the storage of numerical data, which can be affected by the program. In the second statement there is a slight misconception; data is "moved", rather than the instructions being atomic. This may, of course, be due to language semantics, rather than a misunderstanding.

The users also connect a register machine to the function of a computer; "simple, virtual computer" and "basic CPU", proving the tutorial has linked these two concepts. The first also mentions that the register cannot hold negative integers. It would be interesting to know if they understand the impact of this. The tutorial does explain how CPU works with just binary,

and it is the programs which interpret the data. It would be an advanced step if the user were able to apply this concept to Reggie, having a register hold a sign flag.

In hindsight, a further question asking how a computer uses the data in memory would have shown if the users understood that simple, atomic calculations performed by the CPU on the binary data are the basis for every single function and process of a computer, and that interpretation of the data is solely down to the overlying program or, in the case of Reggie, the user.

One user did have prior knowledge of CPU function, as previously mentioned, and their response to questions about CPU functionality did not differ before and after use of Reggie. The user with no knowledge of CPU function, when asked to describe how a CPU completes a function stated:

"The CPU processed instructions and directs the register. It carrys out millions of calculations per second. The CPU stores numbers in memory in binary and carry out calculations after reading them from memory."

They were then asked to compare a CPU with register machine; *"The CPU gets a command and stores the command in memory (registers)."*

Overall, their knowledge has increased; they understand that a CPU processes instructions, performing calculations on binary, which is stored in memory. They know that the CPU stores commands in memory, as well as the operands for the instructions. Although they did not fully answer the comparison question, they had previously stated that a register machine simulator is a "simple virtual computer", showing they do understand the connection.

It is unclear what they meant by "directs the register"; it could be interpreted that the instructions direct the CPU where to store data, because the registers hold the data in Reggie, which is correct. It could also be read to mean that the CPU directs the register to complete calculations, which is incorrect.

A final, particularly positive outcome was this comment:

"Reggie was fun to use, I plan to keep using him after this to figure out some ideas I have for more programs – I WILL master multiplication eventually!"

Despite having no gamification elements, this user is still motivated to return, and found the application fun to use.

Even with the small sample size, it did appear that Reggie helped convey the concept of a register machine, and connect this with CPU functionality. A more in depth questionnaire with a larger sample size would provide a better indication.

Testing

Throughout the development, as new code was completed, informal unit testing was completed on each method. This included white-box testing, following the path of data through the methods, and black-box testing, where the author interacted directly with the application, keeping note of any bugs, and updating them. Towards the end of the project, formal testing was completed. In some cases JUnit testing was completed, and in other cases, black box testing was more appropriate. Due to time constraints, the formal testing was not as in depth as would be expected of an application to be deployed. However, it is believed that good coverage was achieved by the regular informal testing, wide ranging user evaluation, and the series of formal, documented tests which follow. In general, these tests are unit, state tests; testing the methods, and the results of the methods.

There is a testing framework for Android. This API is based on JUnit 3, and has been extended to include Android specific testing. For example, the life cycle of an Activity can be tested, ensuring it is created and destroyed correctly and, as well as the standard Assert.class methods, Android has specific classes for testing View placement (ViewAssert.class). (reference).

This framework must be used for any Android project but, where a class does not contain Android specific elements, like Views and Activities, standard JUnit code can be used.

MainActivity.class had to use the Android framework, CreateInstruction.class

Game.class, RunGame.class

The Instructions were tested using standard JUnit code.

ErrorCheck.class

Testing of SaveLoad.class was black box. This is due to the previous testing of MainActivity.class and Game.class, which had proven both the correct visualisation of the program, and running the program, from the list of instructions. The error checking prior to file creation is completed by ErrorCheck.class, also previously tested.

The files saved by this class take the program from Game.class, and create a file written in Dennett's RAP language. Files of the same structure can also be loaded into the application.

Four programs were created (Add, Move, Copy and Subtract with a flag to indicate if the answer is a minus). These were saved as .txt format by the application. The data in these files

was manually confirmed to be correct; in Dennett's RAP language, specifying the correct operations.

The files were then loaded back into the application. Screenshots from before and after confirmed the correct program was displayed on screen. The programs performed correctly, as was expected from MainActivity.class and Game.class testing.

Next, three malformed files were created to ensure all error messages were firing correctly. These were loaded into the application, and the error messages checked.

An attempt was made to create malformed programs on screen, to determine how SaveLoad.class would cope with this for file output. It was decided to create more error checks on programs, rather than allow a user to export a program with errors such as multiple arrows coming from the same Box. As the application is an educational tool, it should direct users to create correct programs, as they may not have the knowledge to debug without help from the application.

DrawArrow.class was black box tested, as the results of the class are displayed on the device screen. Arrow.class has previously been tested, and it has been shown that it calculates the correct span. The ImageButton Sizer, providing the width and height, has been tested within MainActivity.class; therefore, the most appropriate way to judge if the DrawArrow.class is working correctly is to verify the visualisation on screen. The size, placement and shape of Arrows were verified as correct during the creation of multiple simple and complex programs. Examples of this are seen in (Program Figure), and (FIGURE SHOWING COMPLEX SUBTRACTION).

The Tutorial.class, because of its relative simplicity, displaying sequential Dialog and ShowcaseView, was tested by the author and evaluators interacting with it numerous times. Its actions cannot be changed by the user; the class holds its position, and increments this, showing the appropriate Dialog or ShowcaseView. Due to this, testing only needed to consist of taking the expected displayed and comparing this with the actual step displayed. This verified the correct working of the class. Any complexity, for example, the loading of the program, is handled by Game.class or SaveLoad.class, both previously tested.

OptionScreen.class and SplashScreen.class, due to their trivial nature, were not formally tested; they are verified as working correctly each time the application opens. `TwoDScrollView.class`, aFileDialog and ShowcaseView were also not tested, as this is outside the scope of the project. These open source projects were assumed to be working correctly. Their limitations have been documented throughout this dissertation.

Conclusions

This project had one main aim; to deploy RodRego on a modern platform. The development of Reggie, an Android application, succeeded in this. The application is a register machine simulator, mirroring the actions of RodRego. Programs can be built, run and deleted. The system status is clear at all times; the effects on register data are highlighted as steps in a program are completed.

Of course there were other essential requirements for the developed application to be considered a success, as RodRego is an educational tool. Reggie has satisfied the majority of these. It contains a tutorial that is considered clear and conveys the concept of a register machine. The interface is interactive, dynamic and encourages users to actually play with the application, whilst reducing the possibility of user error; unlike RodRego, the user can only select predefined instructions and registers, and the error messages are parameterised, with additional assistance to fix the issue. The visualisation of programs was appropriately designed; all users are able to follow the progress of a program running, as well as determining which register an instruction will impact, and most can decipher the future process of the program prior to seeing it run.

The main extensibility of the application is the file input/output capabilities; this allows users to create programs in Dennett's RAP language and load these. An interesting aside to this is ability to easily share programs that have been created.

The success of the learning outcomes, particularly about CPU functionality, are not as clear cut; users definitely had an increased knowledge, but still had some misconceptions and issues. This, however, could not be considered a failure; educational tools should provide a springboard, a basic level of knowledge that assists the user in searching for more information. They cannot possibly provide all of the data required.

Unfortunately, the gamification elements, considered high priority for the project, were not implemented. This was due to issues with creating the Android layout; this risk was detailed in the project proposal, and the mitigation plan of seeking assistance within the department prevented the project from becoming infeasible, but the delay in resolution caused time constraints.

The outstanding issues, aside from the gamification, centre mainly on the layout, with some within the code design; these are detailed through this dissertation. A reimplementation of the interface as a custom View would resolve some of the drag and scroll issues, as well as making the app accessible on all device screen sizes. Refactoring the code using the Visitor pattern would reduce coupling between classes, and being able to sort instructions would make program file creation more efficient. As the biggest frustration for the users remains the arrow visualisation, and interaction with it, the dragging operation needs updating to capture the velocity and direction of the dragging action.

Future Work

Possibilities for future work, and how these would be implemented, have been detailed previously; adding translated text and larger icons to the /res directory would expand accessibility to larger screens, and other languages, running a program fast or slow, and providing a “Back” button on the tutorial.

There were other requirements that were not implemented, as they were aimed at more advanced users; subroutines, comments, breakpoints and additional registers. Some of these are simple to implement. Allowing a user to select the number of registers would merely be a case of increasing the number of available colours in the /res directory, with matching colour icons. The constant of maximum register allow is set in only one place, `Game.class`, and could be updated by a user entered number. Comments for instructions would require a new instance variable for each instruction to hold the `String` comment and new user event listener methods in `MainActivity.class`; a long press to view/edit the comments would be appropriate. The `AlertDialog.class` in Android can accept an XML layout and, so, a text field could be displayed to the user when they long press to capture the comment and display it. Breakpoints could be implemented at the same time; the `AlertDialog.class` that appears when the instruction is long pressed could have a checkbox for “Stop when this instruction is reached”. The `RunGame.class` would check a `boolean` value, set by this checkbox, before running an instruction.

The help mode suggested by many evaluators was the most interesting new requirement. It would be simple to implement, and would add a lot of value to the application for novice users.

Finally, not considered for this project, but some thought-providing future work is how best to display a program; if a user has redundant, or unused instructions, could Reggie suggest a more efficient, or better way to design the program?

Appendix X

- Must run on a modern platform
- Must be educational

  Must maintain and convey register machine concept

  Must link register machine concept with CPU functionality

  Must have tutorial/help
- Must be possible to create a program
- Must be possible to visualise the program on screen

  Instructions must be visually differentiated

  Instructions must be visually linked to registers
- Must be possible to run a program, visualising the effect on register data

  Program must run as expected, each instruction actioned in turn.
- Must be possible to delete a program from screen

- Should be usable by naive user

  Should provide clear, parameterised error messages

  - Error messages should contain the instruction with the issue
  - Error messages should have steps to resolve the error

  Should reduce the chance of user error

  - User should not be able to complete incorrect actions
    - User should select instructions from predefined list
    - User should be warned if instructions are not correct (a decrement without a branch, for example)
  - Actions of buttons should be clear
    - Should use standard icons where possible
    - Should label buttons where applicable

  Should contain an in depth tutorial/help

  - Tutorial/help should contain text and pictures
  - Tutorial/help should detail all steps to create a program
  - Tutorial/help should not contain technical terms

Should allow for undo of action

System status should be clear to user at all times

- Status of "Running"
- Status of "Not Running"
- Affected registers
- Current instruction being completed

- Should allow save of program to a file, and loading of program from a file.

  File should have defined structure

  File should be widely used format

  File should be editable outside application

  File should not import if incorrect format

  File should not export if program has errors

  File should output in Dennett's RAP language

- Should challenge users to complete tasks

  Should keep track of completed challenges

  Should provide motivational feedback

- Should have interactive interface

  User should be able to create, edit, run and delete a program

- Should be accessible

  Should use colours which can be discerned by the colour blind

- Could allow for specifying subroutines.
- Could allow comments to be added to instructions.
- Could allow for breakpoints to be added.
- Could allow for additional registers to be added.

- Would be implemented across multiple devices.

Tool would be flexible and automatically scale for larger/smaller screens.

References

Android. (No date). Using Hardware Devices. [Online]. [Accessed 26Mar2014]. Available at: http://developer.android.com/tools/device.html

Apple. (2014). iOS Developer Program. [Online]. [Accessed 26Mar2014]. Available at: https://developer.apple.com/programs/ios/

Baddeley, A. D., & Longman, D. J. A. (1978). The influence of length and frequency of training session on the rate of learning to type. *Ergonomics*, **21**(8), pp627-635

Bird, S. (2010). Effects of distributed practice on the acquisition of second language English syntax. *Applied Psycholinguistics*, **31**(4), pp635-650.

Bort, J. (2013). Google: There Are 900 Million Android Devices Activated. *Business Insider.* [Online]. 15May. [Accessed: 26Mar2014]. Available at: http://www.businessinsider.com/900-million-android-devices-in-2013-2013-5

Brasell, H. (1987). The effect of real- time laboratory graphing on learning graphic representations of distance and velocity. *Journal of research in science teaching*, **24**(4), pp385-395

Brown, J. S., Collins, A., & Duguid, P. (1989). Situated cognition and the culture of learning. *Educational researcher*, **18**(1), pp32-42

Carpenter, S. K., Pashler, H., & Cepeda, N. J. (2009). Using tests to enhance 8th grade students' retention of US history facts. *Applied Cognitive Psychology*, **23**(6), pp760-771

Cepeda, N. J., Pashler, H., Vul, E., Wixted, J. T., & Rohrer, D. (2006). Distributed practice in verbal recall tasks: A review and quantitative synthesis. *Psychological bulletin*, **132**(3), pp354

Chen, G. D., Chang, C. K., & Wang, C. Y. (2008). Ubiquitous learning website: Scaffold learners by mobile devices with information-aware techniques. *Computers & Education*, **50**(1), pp77-90

Chin, E., Felt, A. P., Sekar, V., & Wagner, D. (2012). Measuring user confidence in smartphone security and privacy. In Proceedings of the Eighth Symposium on Usable Privacy and Security (p. 1). ACM

Clark, R. C., & Mayer, R. E. (2011). *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. John Wiley & Sons. pp133 – 142

Colour Blind Awareness. (No Date). What is colour blindness? [Online]. [Accessed 26Mar2014]. Available at: http://www.colourblindawareness.org/colour-blindness

CTIA. 2012. CTIA's Semi-Annual Wireless Industry Survey Results. [Online]. [Accessed: 26Mar2014]. Available at: *http://files.ctia.org/pdf/CTIA_Survey_YE_2012_Graphics-FINAL.pdf*

De Bortoli, A. (2010). IURM (Unlimited Register Machine). [Online]. [Accessed: 26Mar2014]. Available at: https://itunes.apple.com/us/app/iurm/id386981571?mt=8

Dennet, D.C. (No Date). RodRego - Register Machine Simulation (Ver 1.3). [Online]. [Accessed: 26Mar2014]. Available at: http://sites.tufts.edu/rodrego/

Dennet, D.C. (2008). The Secrets of Computer Power Revealed. [Online]. [Accessed: 26Mar2014]. Available at: http://sites.tufts.edu/rodrego/files/2011/03/Secrets-of-Computer-Power-Revealed-2008.pdf

Eshach, H. (2007). Bridging in-school and out-of-school learning: Formal, non-formal, and informal education. *Journal of Science Education and Technology*,**16**(2), pp171-190

Falk, J. H. (1983). Field trips: A look at environmental effects on learning. *Journal of Biological Education*, **17**(2), pp137-142

Fallahkhair, S., Pemberton, L., & Masthoff, J. (2004). A dual device scenario for informal language learning: interactive television meets the mobile phone. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies* (pp. 16-20). IEEE Computer Society

Finson, K. D., & Enochs, L. G. (1987). Student attitudes toward science- technology- society resulting from visitation to a science- technology museum. *Journal of Research in Science Teaching*, **24**(7), pp593-609

Gerber, B. L., Cavallo, A. M., & Marek, E. A. (2001). Relationships among informal learning environments, teaching procedures and scientific reasoning ability. *International Journal of Science Education*, **23**(5), pp535-549

Harrison, R., Flood, D., & Duce, D. (2013). Usability of mobile applications: literature review and rationale for a new usability model. *Journal of Interaction Science*, **1**(1), pp1-16.

Harvey, H. W. (1951). An experimental study of the effect of field trips upon the development of scientific attitudes in a ninth grade general science class. *Science Education*, **35**(5), pp242-248

Issroff, K., Scanlon, E., & Jones, A. (2007). Affect and mobile technologies: case studies. In *BEYOND MOBILE LEARNING WORKSHOP* (p. 18).

Kalyuga, S., Chandler, P., & Sweller, J. (1999). Managing split-attention and redundancy in multimedia instruction. *Applied cognitive psychology*, **13(**4), pp351-371

Lam-Kan, K. S. (1985). The contributions of enrichment activities towards science interest and science achievement. Unpublished master's thesis, National University of Singapore, Singapore. [ABSTRACT ONLY]

Lampert, M. (1990). When the problem is not the question and the solution is not the answer: Mathematical knowing and teaching. *American educational research journal*, **27**(1), pp29-63

Mocker, D. W., & Spear, G. E. (1982). Lifelong Learning: Formal, Nonformal, Informal, and Self-Directed. Information Series No. 241

Mumtaz, S. (2001). Children's enjoyment and perception of computer use in the home and the school. *Computers & Education*, **36**(4), pp347-362

Najjar, L. J. (1998). Principles of educational multimedia user interface design. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, **40**(2), pp311-323

Nielsen, J. (1995). 10 Usability Heuristics for User Interface Design. [Online]. [Accessed 26Mar2014]. Available at: http://www.nngroup.com/articles/ten-usability-heuristics/

Pappano, L. (2012). The Year of the MOOC. *The New York Times.* [Online]. 2Nov. [Accessed: 25Mar2014]. Available at: http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html?_r=0

Reisner, E. R., White, R. N., Russell, C. A., & Birmingham, J. (2004). Building quality, scale and effectiveness in after-school programs. *Washington, DC: Policy Studies Associates*

Rogowsky, M. (2014). Without Much Fanfare, Apple Has Sold Its 500 Millionth iPhone. *Forbes.* [Online]. 25Mar. [Accessed 26Mar2014]. Available at: http://www.forbes.com/sites/markrogowsky/2014/03/25/without-much-fanfare-apple-has-sold-its-500-millionth-iphone/

Rohrer, D., & Taylor, K. (2006). The effects of overlearning and distributed practise on the retention of mathematics knowledge. *Applied Cognitive Psychology*, **20**(9), pp1209-1224

Russell, J. W., Kozma, R. B., Jones, T., Wykoff, J., Marx, N. & Davis, J. (1997). Use of simultaneous-synchronised macroscopic, microscopic and symbolic representations to enhance the teaching and learning of chemical concepts. Journal of Chemical Education, **74**(3), pp330-334

Safe Web Colours (No Date). Safe Web Colours. [Online]. [Accessed 26Mar2014]. Available at: http://safecolours.rigdenage.com/index.html

Schulte, C., & Knobelsdorf, M. (2007). Attitudes towards computer science-computing experiences as a starting point and barrier to computer science. In *Proceedings of the third international workshop on Computing education research* (pp. 27-38). ACM.

Stockwell, G. (2010). Using mobile phones for vocabulary activities: Examining the effect of the platform. *Language Learning & Technology*, **14**(2), pp95-110

Sutherland, R., Facer, K., Furlong, R., & Furlong, J. (2000). A new environment for education? The computer in the home. *Computers & Education*, **34**(3), pp195-212

Szabo, M., & Kanuka, H. (1999). Effects of violating screen design principles of balance, unity, and focus on recall learning, study time, and completion rates. *Journal of Educational Multimedia and Hypermedia*, **8**(1), pp23-42

Thornton, P., & Houser, C. (2005). Using mobile phones in English education in Japan. *Journal of computer assisted learning*, **21**(3), pp217-228.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, **58**, pp345-363

Vadeboncoeur, J. A., Blackburn, M., & Moje, E. B. (2006). Engaging young people: Learning in informal contexts. *Review of Research in Education*, pp30, 239

Vali, A. (2007). Storytelling: An Effective Method for Teaching Complicated Concepts in Medicine. *Iranian Journal of Medical Education*, **7**(1), pp155-160. [ABSTRACT ONLY]

Vischeck. (No date). Color blind image correction. [Online]. [Accessed 26Mar2014]. Available at: http://www.vischeck.com/daltonize/

Vogel, J. J., Vogel, D. S., Cannon-Bowers, J., Bowers, C. A., Muse, K., & Wright, M. (2006). Computer gaming and interactive simulations for learning: A meta-analysis. *Journal of Educational Computing Research*, **34**(3), pp229-243.

Wang, H. (1957). A variant to Turing's theory of computing machines. *Journal of the ACM (JACM)*, **4**(1), pp63-92

Wishart, J. (1990). Cognitive factors related to user involvement with computers and their effects upon learning from an educational computer game. *Computers & Education*, **15**(1), pp145-150