Developing high-fidelity mental models of programming concepts using manipulatives and interactive metaphors.

Submitted in partial fulfilment
of the requirements of the degree of

MASTER OF SCIENCE

of Rhodes University

Supervisor: Professor Peter Wentworth.

Grahamstown, South Africa
30/10/2014

Abstract

This only gets done at the very end, once we have a conclusion that can be included here.

Acknowledgements

First and foremost I would like to thank my fiance for her continual support while working on this project, especially when I felt like I simply could not keep going. I would also like to thank my supervisor for his valuable advise and opinions. Finally I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

Chapter 1

Introduction

> "It was found that students with viable mental models performed significantly better in the course examination and programing tasks than those with non-viable mental models." Ma et al. [1]

This piece of information alone is enough to warrant investigation into what a viable mental model is, how they are developed normally, and how they can be made more viable. Ma et al. [1] go on to say that "students must be supported to create new viable models", which is a problem that this project aims to do address.

<THIS IS ONLY VERY ROUGH TRAIN OF THOUGHT WRITING:>

As a tutor and a once novice programmer, I've noticed that there are several conceptual hurdles that beginner programmers struggle to overcome. In some situations the conceptual models that a student builds to overcome such hurdles have to later be modified or completely replaced because the model they came up with was flawed. It appears as if this process over of laboured mental model development, correction, and replacement is an unnessary obstacle to the learning of programming.

This thesis aims to overcome said hurdles through the creation of several high-fidelity metaphors, and the implementation of said metaphors as interactive manipulatives.

The general goals for this paper include:

- Identification of troublesome concepts.
- A brief survey of existing metaphors.

- The creation of a set of high fidelity metaphors.
- An interactive implementation of said metaphors.
- A brief survey and comparison of existing games that teach programming.
- Careful testing of the usefulness of the metaphors.

An excellent

Chapter 2

Literature Review

2.1 Preliminary Definitions

For the sake of clarity it is necessary to define certain terms such as manipulative and metaphor, in the context of this project.

2.2 Similar Work.

What work like mine has been done?

2.3 Education theory and Learning styles

2.3.1 Constructivism

2.3.2 Kolb and Active experimentation

David A. Kolb is an American educational theorist, whose underlying philosophy is one of constructivism. Explain various learning styles (such as those by David Kolb) and how my work fits in with it/them as "Active Experimentation",

https://en.wikipedia.org/wiki/Learning_styles

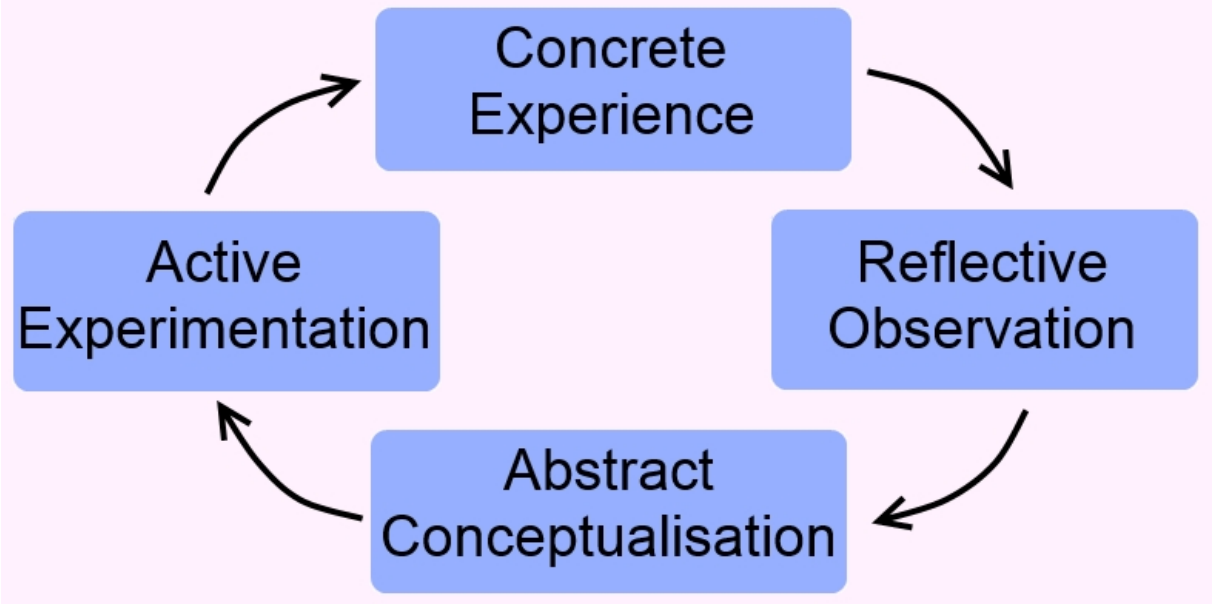

Figure 2.1: The Kolb Learning Cycle

http://infed.org/mobi/david-a-kolb-on-experiential-learning/

http://www2.le.ac.uk/departments/gradschool/training/eresources/teaching/theories/kolb

2.3.3 Montessori Education

<Explain Montessori education in general>

<This probably paragraph probably belongs in the discussion section>Mostly used for kids, they are encouraged to experiment with various manipulatives of their choice with a trained teacher there to guide them. There is a parallel between this and the metaphor game that was created for this project: both use manipulative, the game uses carefully constructed help screens and hint systems to take the place of the teacher, and finally students have a choice of activity in so far as they can choose which constructs and concepts they would most like to practice.

https://en.wikipedia.org/wiki/Montessori_education

Concept/Structure	Agreeing authors	Qualifies as a threshold concept
Method Calling		
Value Parameters		
Reference Parameters		
Return Values		
Objects		
Strings		
Arrays and Lists		
Assignment		
Recursion		
Events		
State Machines		
Conditionals		
Iteration/Loops		
Variables		
Assignment		
Types		
User defined data types		
Pre and post conditions?		

Table 2.1: Assorted concepts

2.4 Common Misunderstandings, and troublesome concepts.

A list of concepts, some of which are cited as being more troublesome than others:

A study done by Bayman and Mayer[2] found that a group of self taught BASIC students had trouble with even very simple statements, most if the tested statements were understood by less than 43% of the students in the study. Further examples include: only 3% understood what 'input A' meant, while 27% understood "IF A < B GOTO 99". While BASIC might be less verbose about the meaning of each line when compared to something like C# or Java, the difference is not so extreme as to render this work irrelevant. The major idea that one can take away from this study is that without the guidance of an experienced programmer, students will often have little to zero understanding of the language constructs, not to mention more complicated concepts such as mutable vs immutable, or passing by reference rather than value.

A secondary point that can be taken away from Bayman and Mayer [2] is that even the simplest of statements could benefit from more detailed explanations (which is where my metaphors come in).

Algorithm 2.1 Simple assignment statements

```
int x = 2;
int y = x;
x = x + 5;
Console.WriteLine(“x -> {0}, y -> {1}”, x, y);
```

In a more recent study on the understanding of recursion Götschi et al.[3] found that on average less than 50% of first year students held viable mental models of recursion. <Theres more to say about recusion>.

Personal experience has demonstrated that some novice programmers will treat a series of assignment statements more like simultaneous equations that a series of ordered commands, for example students see line 2 in algorithm 2.1 and assume that x and y are now one and the same, thus their final output is “x -> 7, y -> 7” rather than “x -> 7, y -> 2”. <find another author that demonstrates this>.

2.4.1 Threshold concepts

Roundtree and Roundtree [4] describe concepts as being either core concepts, threshold concepts, or both. They describe a threshold concept as being an idea or way of thinking that is key to not only understanding a subject but to beginning to think like a practitioner of said subject. For example simply understanding how a computer scientist thinks is not the same as being able to think like a computer scientist. <There is more to be said about the work by these 2>

2.5 Mental models

In their study of students learning recursion Götschi et al.[3] identified at least 7 distinct non-viable mental models for recursion. Some of the mental models they identified are as follow: some students applied mathematical mental models to the recurrence relationship, while others applied a “return-value mental model” [1]. According to Götschi et al.[3] the return value mental model is probably due to students not having a “viable model of parameter passing and return value evaluation which are required before students can understand how a recursive program executes”. What this means is that when students

[1] In this mental model recursion stops at the base case and the final value is returned immediately

have low-fidelity models for more elementary concepts, they try to either maintain or adjust their existing mental models to suit the new concept and end up misunderstanding. This is an excellent reason why even the most elementary concepts need to have high fidelity models from the start.

Götschi et al.[3] goes on to explain that some students hold multiple *viable* mental models and use them selectively depending on the question. <Further discussion of how this work relates to my work and mental models belongs in discussion>

<Find other authors that have studied mental models>

2.6 Established metaphors and manipulatives

List and explain all of them, discussion and opinions/comparisons are to be kept until the discussion section.

- Local Variables
  - Assignment
  - Reading
- Arithmetic and Expressions
- Methods
  - Parameters
    - Pass by value
    - Pass by reference
  - Returning and return types
- Strings
- Arrays
- Classes
- Constructors
- Objects

2.7 Summary of the Literature Reviewed

A brief summary of what the literature reviewed says.

Chapter 3

Discussion

This discussion will address several issues brought up in the literature review. Additionally it will compare and contrast what various other academics have said. Highlight things that are lacking or conflicting in the literature, personal opinions (based on logic) can come in here.

3.1 Determining success

Refer to personal correspondence with Pete (13 Feb 2014) for all the details (and other sources) about Corridor testing and how exam questions are designed to test a particular section of a students understanding, and therefore how making levels based on exam questions is also a valid means of testing understanding and gauging success. If a student couldn't understand a particular concept before using the program, but could after using a level designed to target that area, there is a good chance the program succeeded in helping. Some of this belongs in the literature review.

Refer to appendix X to see a more complete list of the various exam questions and their corresponding level code.

3.2 Troublesome concepts

When examining the results of a 2013 mid-year Exam these are the programming questions that people did most poorly on:

Question	Correct answers
If A=true, B=false, C=5, D=3, then A || C != C && D + 2 <= C is...	63%
The opposite of > is...	64%
Where in the code does the declaration of a class attribute belong.	71%

...So that exam was probably too easy with most questions being correctly answered by over 71% of the class.

This table is for poorly answered **programming** questions in the final exam of the same year:

Question	Correct answers
A class with a static and a none static attribute. 2 instances. Output?	52%
Test if students know that strings are immutable.	53%
The meaning of internal as an access modifier	53%
Overwriting reference types after passing as parameters.	56%
Exaplin the meaning of IntelliSense doc <does it apply here?>	61%
Default access modifier for method?	66%
Differences between lists and arrays	71%

With an MCQ average mark of 70% this exam might also have been too easy...

These results alone are not enough to go on in order to draw conclusions about what are some common troublesome concepts. Therefore we need to get additional data from other sources.

3.3 Alternative representations for a single concept.

<this is not formally written yet:>

As an example of something that has an alternative representation, the concept that came up most frequently was that of the values declared variable have before they are assigned. In this project the guideline used for determining this sort of thing was “how it it represented in the Visual Studio debugger”, in this particular case however we found something of a dark corner to the debugger: If you set a break point on a variable that has been declared but not assigned, and you try to read the variables value, Visual Studio will still display a value (defaulting to 0, 0.0, null, or whatever is the most appropriate). This

is inconsistent with what students expect, and how we would like them to understand variables, in that it makes variables *appear* as though they can be used before they are assigned. An example of code that could confuse students might be:

```
int x;
int y = 0;
```

By the end of this example, if we were to keep the technique used in VS, there would be no distinguishing between the two variables in game environment. A number of possible representations were put forth:

- Variables behave just in VS - a default value that is then displayed openly.
- Variables take their initial value from the hand, when declared.
- In the back-end variables would get given a default value that would match VS, except the variable would be shown as entirely empty on the front-end.
- In the back-end variables would get given a default value that would match VS, except the variable would be shown as containing "???" in the front-end.

There are pros and cons to each of the proposed techniques, for example leaving the variables blank on the front end would drive home the fact that you can not read from it yet, but a potential downside would be students thinking that variables could be entirely empty (and nothing on a computer is ever truly empty). Initially it was decided that variable would take their initial value from the hand, however as explained later in section 6.1, students found this to be confusing in a number of ways. We finally settled on applying a standard default value in the back-end, but displaying "???" to the users until they had assigned a value.

Chapter 4

Methodology

This section explains the methods and techniques we used for this project.

4.1 Design based research

<theres a lot that can be said here and you should look for further references, for now we have a little bit of intro info:>

Juuti and Lavonen[5] explain that design based research is defined by three major attributes:

1. The primary goal of the research being performed is to create an artefact that assists both instructors and learners.
2. Design-based research proposes or produces original theories regarding education and learning theory.
3. The entire process is iterative.

According to Juuti and Lavonen[5] only when these three attributes are combined can one consider their research design-based.

4.2 MVC build

Due to initial uncertainty around which metaphors complement each other best, while retaining fidelity and encouraging consistent mental models, an MVC approach was decided upon for implementing the game. An MVC approach allowed us to build the underlying API for tracking state and fundamental logic (model), separate from the visual metaphors and manipulatives that the end user is presented with (view). The model and the view then come together in through the controller.

This design paradigm allows us to easily switch out components on either the front end or the back end without any side effects. Parts of the underlying API as well as the displayed metaphors, needed to be switched out and improved upon, especially once usability testing began (as explained in section 4.4). Fortunately changes to one side seldom eaffected the other. An example of a change to the model section would be when the order users needed to perform actions was altered (a common example of this was in regard to variable declaration and assignment as explained in section 3.3). View alterations occurred whenever a user didn't understand something as expected, or when feedback was given about the interface lacking something (or being unintuitive).

Implementing a system such as this without using MVC as the underlying design paradigm would be unreasonably time-consuming simply due to the frequency with which changes are made. Using the MVC paradigm not only makes development easier, it also makes the underlying API more portable, thus allowing other users to use the API as it is without any changes. <Consider including a small diagram to illustrate the MVC nature of the project:underlying classes - world tracker - game1 - visual classes>

4.3 A new metaphor set

This section describes the complete finished metaphor set after optimisations were made, as well as the original metaphors before optimisations were made, and finally some alternative representations and potential enhancements. The metaphors explained are not the ones originaly presented to students, section X describes the iterative development of the metaphors in more detail. When students seemed to not grasp a particular metaphor during usability testing (as explained in section X), the troublesome concept and associated metaphor were considered carefully and altered to address the issue, and it is the improved metaphors that are described here. Section X contains more details about what students students struggled with and how the issues where addressed.

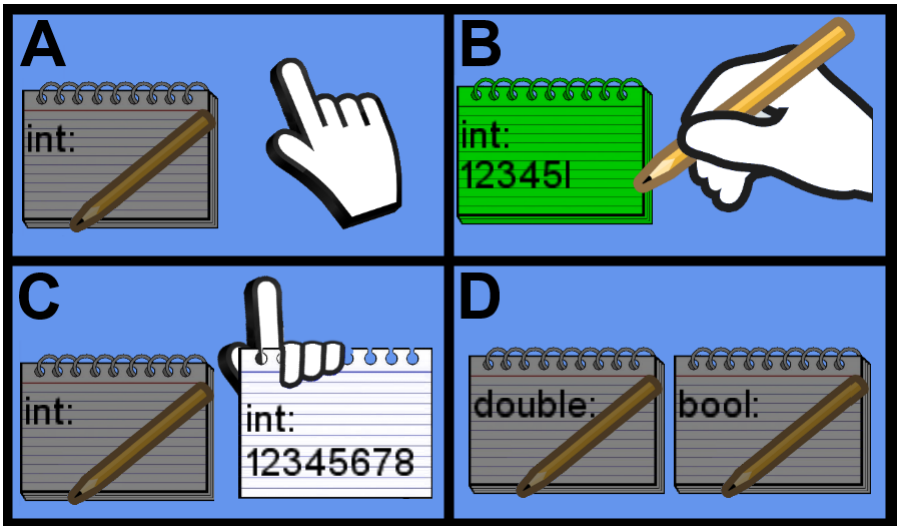

Figure 4.1: A: An int value-notepad. B: User inputting a value. C: The user holding the final value. D: Alternative value-notepads.

4.3.1 Values

The most fundamental things that code uses are individual values, either as actual values or the value of a reference address (this would include integers, doubles, booleans, chars, and all other primitives). As this is such a key concept, its associates metaphor needed to be one of the most reliable and easy to understand: it was concluded that a simple piece of paper with the value written on it would be suitable. Everyone can relate to pen and paper, you cannot erase pen from paper (meaning values are never 're-used'), and if one were to imagine solving an expression in their head the most sensible thing to do with the answer would be to write it down. This representation of values laid the foundation for the remaining metaphors.Figure 4.1 shows an example of a value notepad, the user inputting a value, and finally the user holding the value.

4.3.2 Variables

Variables are represented by boxes, with transparent lids, that contain a single piece of paper (in the same way that a simple variable can contain a single value). Reading of a variable would be done courtesy of the transparent lid, you would simply look into the variable box and copy the value off the paper without changing the content of the box. Assigning to a variable involves opening the box, disposing of the old value-paper, and then placing the new value which you would be holding into the box. The metaphors for both values and variables can be extended to include type limitations: one potential way

Figure 4.2: A variable box called counter, containing the integer value 12345678.

to do this would be to use different coloured paper and boxes for each type, or alternatively you could have boxes and pieces of paper that are of different shapes and sizes, so that one type cannot fit into another type, finally you could make users use different pens to write down differently typed value. These potential extensions do not interfere with the more advanced metaphors. Figure 4.2 shows one potential version of the variable box metaphor.

4.3.3 Expressions, arithmetic and calculation.

<Expand on the Arithmetic and Expressions and the calculator> Expressions of all kinds have two viable metaphors in our unified set: they can either go through an in-game calculator, or the user can work them out in their head. Good arguments can be made for either case, and thus both metaphors have been included so that anyone who wants to expand on this work can make up their own mind <expand on the arguments>. The original technique for expressions and evaluation was the use of an in-game calculator, which worked much like a real-world scientific calculator, where user could enter an expression and then go back and alter terms once they had the required values. For example a user might enter "x + 5" into their calculator, they would then look at the local variables and read the value from x as their current value, that value would be fed into the calculator (almost like a fax machine, except the paper be destroyed). When the user asks to substitute into the place holder 'x' in the expression, the calculator would replace the variable in the expression and then wait for further instructions from the user (either for more instructions or for the user to ask it to evaluate the answer). When the expression no longer had place variables that needed values the user would hit 'evaluate' and the answer would be printed out from the calculator onto a piece of paper. Figure X illustrates the sequence of events when using the calculator metaphor.

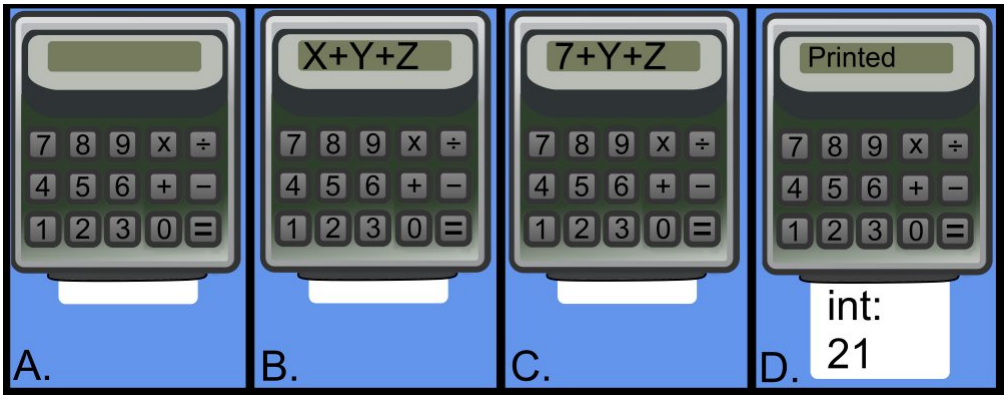

Figure 4.3: The steps required to evaluate an expression using the calculator. A: the calculator is blank and requires an expression. B: An expression has been entered and now requires value-substitutions for every variable. C: A value has been substituted (this requires a variable read first), this step is repeated 3 times. D: All values have been inserted and the user has asked for the answer (which was printed onto a piece of value-paper).

One of the major downsides to this method is the number of steps involved in evaluating an expression, for example 'x + y + z' would require: inputting the expression, three explicit variable reads, three substitutions, and finally evaluation. This number of steps this might not sound like much, but for a beginner any extra steps run the risk of distracting from the code or confusing them, our alternative expression technique needs just one in-game step. A secondary downside to using the calculator as an expression evaluator is that the user wouldn't need **any** understanding of types as the calculator would simply output the correct type along with the answer, this issue could be addressed by altering the metaphor so that users would have to be explicit about the type returned upon evaluation.

An easier technique for expression evaluation actually takes away extra metaphors entirely by just asking the user to evaluate the expression in their heads (or on paper), in much the same way as when users debug a piece of code. This second method poses fewer problems but it is not perfect, the biggest flaw is in fact one that it shares with the calculator metaphor: how to deal with expressions than contain method calls, section X goes into more detail regarding this issue. Steps A, B and C in Figure 4.1 demonstrate how users would enter the answer to an expression.

4.3.4 Local variables and the current stack frame.

The next step up the abstraction ladder is to delimit local and global scope using any kind of uncross-able barrier (for example a line of barbed wire, police tape, or even a fence -

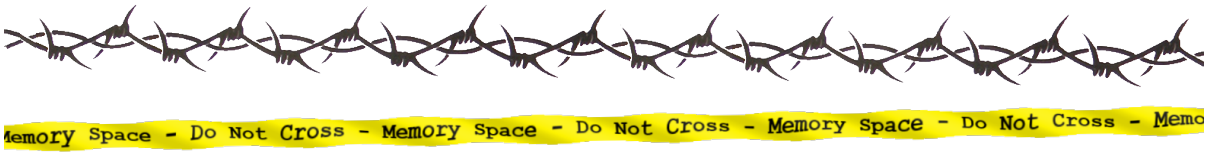

Figure 4.4: Top: Barbed wire memory barrier. Bottom: Police tape memory barrier.

figure 4.4 shows two possible barriers), this barrier serves to separate the user space (i.e. the local scope) from the memory space (i.e. the global scope, including the stack and the heap). The global area will be discussed later in this section, and was mentioned here for clarification of the next concept to described: how to represent the current frame and all the variables that are stored within it, while still being relatable, and without complicating the pushing of stack frames into the global scope.

Several different possibilities were discussed <maybe elaborate on that here>, and finally we settled on something akin to a jigsaw or Lego bookshelf: users would start out without a bookshelf at all when there are no local variables, a mechanism which dispenses bookshelf 'pieces' is all that would be present, whenever the user needs to declare a new variable they would first have to extend the bookshelf so there would be enough space for it. Each space on the bookshelf would be able to contain one variable box. This metaphor might be slightly less relatable than using a normal bookshelf, however a simple bookshelf runs the risk of making the user think the current frame has a fixed size (which is not the case). Another advantage to the expandable bookshelf metaphor is that if you want to teach students about how variables can go out of scope, for example when they were declared inside a loop, the out-of-scope variables (and their associated bookshelf sections) can be removed entirely. The metaphorical bookshelf sits on top of a conveyor belt, this only becomes important when the user is able to call methods, thus it is discussed in more detail in the next section. Figure 4.5 shows what the user might be presented with depending on the current stack frame state. Alternative methods of representing the stack and individual frames where discussed and almost uniformly rejected for various reasons <include examples of alternatives and their shortcomings here>.

4.3.5 The stack and methods

Once one understands how to interpret the representation of the current frame, one is then also able to interpret the stack as a whole: just like a library has more than one bookshelf, a stack has more that one frame - therefore we can cross the two ideas and represent the

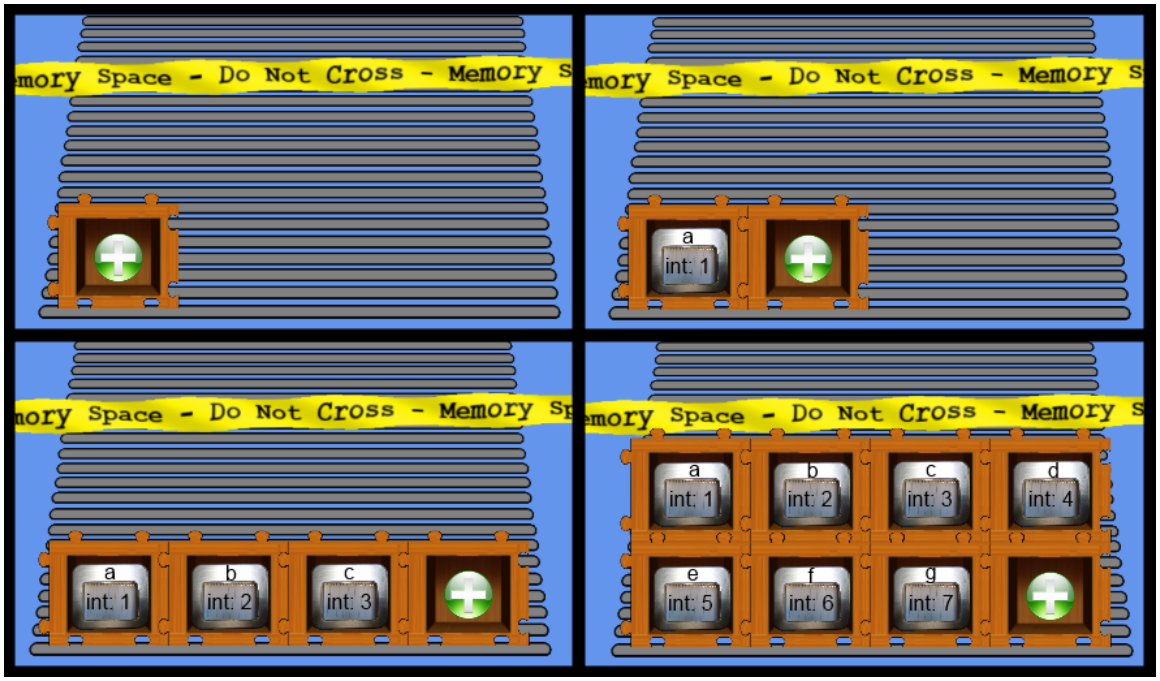

Figure 4.5: The current frame and local variables bookshelf - displayed on top of the frame conveyor belt, and inside the user space. Demonstrating the various states during the declaration of local variables 'a' to 'g'.

stack as row after row of bookshelves. This is also where the division between local and global scope becomes important, as the stack is primarily located in memory space rather than user space. When a new frame is created from a method call, all non-local stack frame bookshelves are moved across the barbed wire divisor so that they exist in memory space (thus allowing for reference variables to be accessed in the same way as objects located on the heap). Figure 4.6 shows how the stack would expand into memory space after each method call, and shrink after each return statement. This in turn leads to the metaphors governing everything to do with methods.

As mentioned in the previous section the current frames associated bookshelf is positioned on top of a conveyor belt, this allows bookshelves to be moved across the barbed wire and into or out of the user space. This mechanism is important for method calls and returns: the conveyor belt has two control buttons, the call button, and the return button. These two buttons move everything on top of the conveyor belt either toward or away from the user space. So far it is not hard to imagine the moving of the conveyor belt as matching up with the actual process of pushing and popping frames to and from the stack; the tricky part of designing this section of the metaphor set is considering with frames that have been popped off the stack, and where do new frames come from. Both situations can can be explained in a similar way to overwritten variable values: For popped frames

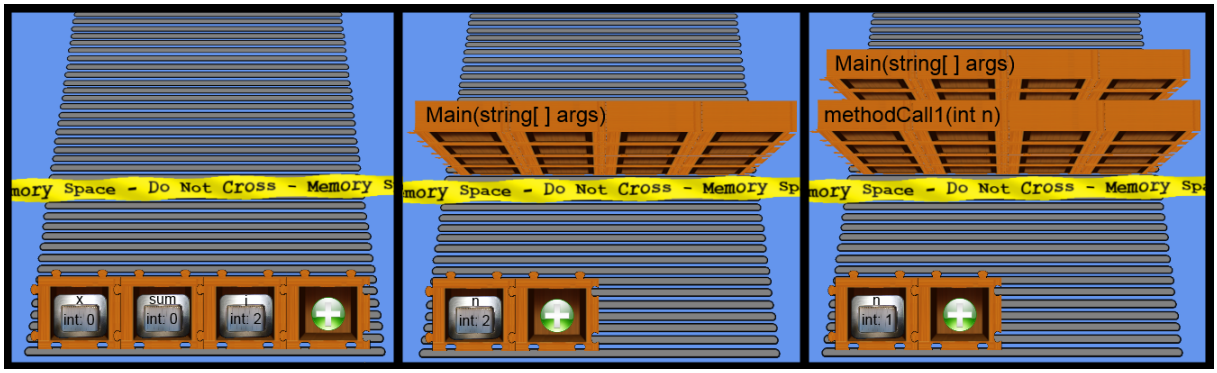

Figure 4.6: Changes in state and visualisation for the stack and the frames that correspond to each method call. Read from left to right this shows the calling of each method, while read from right to left it demonstrates returning back down the stack (pushing and popping respectively).

and overwritten values the object in question no longer belongs anywhere and so must be destroyed, the paper value would be torn up or burn, and a similar thing would be done to the popped off frame. In the same way that we prepare paper values using notepads we decided to use a workbench to represent the creation of a soon to be used frame.

After getting the student to name the method they are preparing to call, you have several options for method preparation that you can present them with, the most feasible two techniques are as follows: the most initially-intuitive way is to ask users to name and assign parameters and arguments in much the same way as local variables (one at a time, with types, name, and values all the explicit responsibility of the user); the second possible presentation method is to give users a selection of available method signatures which they need to pick from, and then provide all the parameters ready to receive values which the user simply slots into place. Figure 4.7 compares the two main method preparation techniques side-by-side. Section X explains how and why we originally used technique one, however later moved to technique two <it was partly due to students trying to do things out of order, being overwhelmed by the lack if information about what to do next, and then that it ran the risk of giving an inaccurate mental model where method signatures where arbitrary or dynamic, in reality all the classes have info about themselves so they know what signature there should be therefore the user should be able to select from the known methods rather than trying to infer from the code>.

For a void method no further explanation is required for how the user leaves the method (they simply press the conveyor belt return button). However it might need saying that a value returning method works by simply making the user hold the return value in their hand, that way the return value is in hand when they get back to the previous frame,

Figure 4.7: Left: Method one for method preparation, where the user is responsible for everything. Right: method 2 of method preparation, where the user is responsible for arguments and calling but nothing more). The stages from top to bottom are: no method named, a method named but no parameters assigned or declared, one parameter (a) named and assigned, all the parameters named and assigned with the method ready for calling.

and thus they can use the value straight away. To some the metaphors governing method behaviour might sound shaky or unclear, however student understanding of this particular subset of metaphors was exceptional, see the usability test results in section X for more details.

4.3.6 Representing the heap

The easiest thing to represent about reference types was what exists in the local scope: a simple memory addresses written down as values. Reference types were represented this way to make it perfectly clear to end users that whatever they do to objects or referenced variables goes through a memory address because the thing in question is not local. Additionally it makes explaining aliasing much easier, for example if the user is presented with code such as:

```
object x = new object();
object y = new object();
x = y;
x.changeSomething();
```

Users can see clearly that initially x and y were different, then y became just another name for x because their values are now the same memory address (and its original value was lost), and then finally changes to one change the other. The more difficult thing to represent about reference types is what gets stored in memory (which the user can partially see over the memory-barrier). Without introducing a middle-man or some other go-between mechanism there is no way for players to affect what exists outside of the user space, to solve this issue we introduce a robot to represent the memory manager, which is elaborated on in section X. For now it is sufficient to say that the memory manager robot receives messages from the user, and carries them out in memory space - taking values from the user and writing them to the heap, or vice versa, depending on the instruction being carried out.

With our memory manager robot ready to interact with the global space on our behalf, we now need a way to represent the heap and access to the stack. Access to the stack is fairly simple as we already have a concrete metaphor for the stack itself (the rows of bookshelves): the memory manager robot simply takes the address he has been given and goes between the bookshelves to interact with the appropriate variable box.

Figure 4.8: The heap bookshelf with two objects in it. If they were array objects they would be of length 4 and 6 respectively. This heap can only contain 36 simple values, and thus demonstrates how a compressed representation would be desirable.

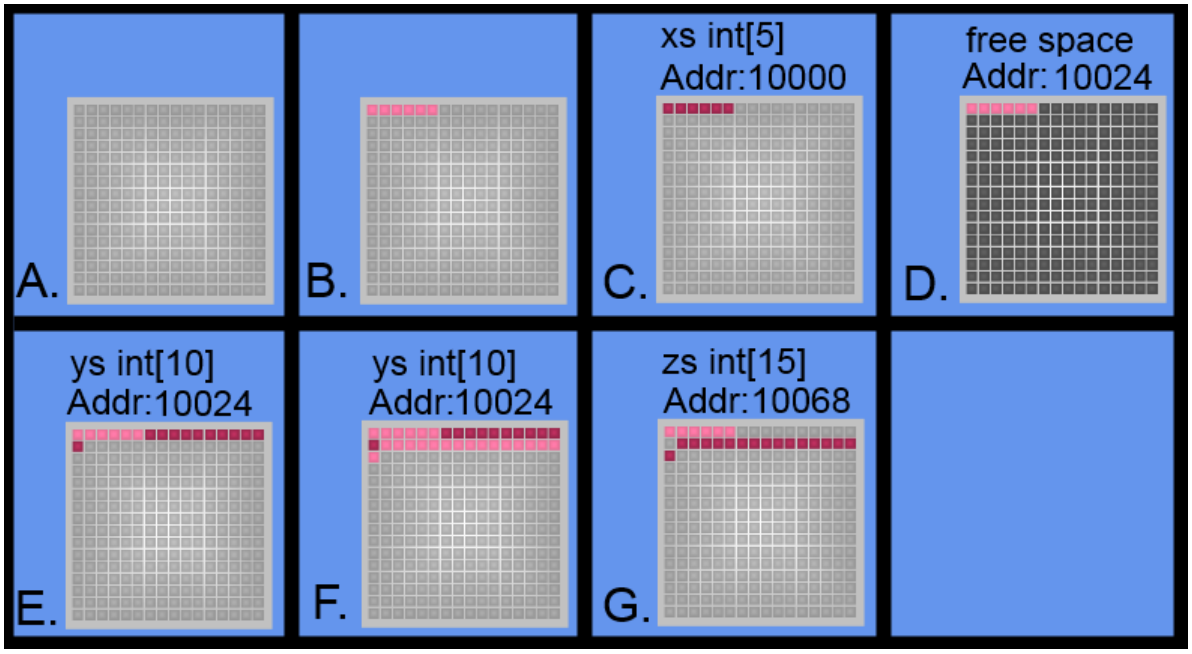

Figure 4.9: A compressed representation of the heap. A: an empty heap. B and C: one array present on the heap, with details above. D: highlighting the free space. E: Three arrays on the heap, with ys being highlighted. G: a demonstration of what might happen after a garbage collection when ys is no longer being used.

The heap requires more thought. After considering that the size of the heap is in fact finite (determined by the available memory) in an actual computer, we decided that the heap metaphor could also be represented by a finite, fixed-size structure. The easiest way to represent it without deviating from our existing metaphors is to have one super-sized bookshelf, possibly with movable dividers, where unused space is represented by the lack of a variable box. Figure 4.8 shows an example of this extra large bookshelf, and also illustrates that in certain media the bookshelf representation falls slightly short due to the size limitations. For this reason, in-game, the heap is instead shown as a compact series of squares (as shown in figure 4.9), this alternative representation can just be thought of a compact version of a bookshelf. It also illustrates that the heap does not need labelling: neither of the objects in the heap has a name shown on the heap, this was done purposely so as to prevent the inaccurate idea that references and spaces in memory have names other than the ones given in the local scope.

There are some disadvantages to this representation, the primary one being that the representation is not based in the real-world and therefore is likely to be less relatable. <explain more about the heap>

Representing the area that a certain object takes up is most easily done using colour.

Figure 4.10: A small section of the heap bookshelf showing potential representations of two non-array objects.

Representing objects, that are more complex than arrays of values, is where reference types become particularly difficult to represent: one cause of this is the wide variety of shapes that objects can take, a secondary issue is that of constructors and when instantiation actually occurs, and a final hurdle is that of objects having their own methods and associated code. <alternate example could be an anonymous bank account> A simple non-array object such as a Random generator could be represented on the heap as two numbers and something to point to the objects associated method code: in this example the numbers would be the start Seed and the previous output (as pseudo random numbers usually rely on previous output), however the methods would have to be seen only in the code that the students are following. Figure 4.10 shows a potential representation of the heap with an instance of the Random and Account classes, unlike figure 4.8 the objects on this heap have their properties labelled so as to make it clear to users that they are not just arrays of numbers. The 'name' property of the Account class would be a string, its contents on the heap would depend on how one chooses to represent strings, as explained in section 4.3.8.

The compressed representation of the heap is not conducive to anything more complex than simple arrays (simply due to labelling limitations), while the more explicit bookshelf representation would allow for objects to have other objects as properties: the object property would simply contain a memory address that points to another object on the heap. As yet neither model has a perfect representation for object code, as strictly speaking each object has it's own code; this needs to be rectified, circumvented, or made very clear before presenting the object metaphors to users.

<Need to explain limitations, the difficulty of using custom classes, and even the trouble inherent when creating simple objects like arrays>

Figure 4.11: The memory manager robot (right) and the terminal used to communicate with him (left). This version of memory space communication was geared towards array only communication.

4.3.7 Interacting with objects and other reference types

<Explain more about the terminal and how the robot work>

Once you understand how the heap and various reference types are represented, communication with them becomes fairly simple to understand as well. The memory manager robot is responsible for all interactions in the memory space, he receives his instruction via the terminal (which is located in local space). The user is able to interact with the terminal, and all values go from user to terminal to robot to memory address (for writes).

These two don't require much explaining, however coming up with them in the first place took several iterations. The same is true of the stack and the heap, section X contains more details about earlier unacceptable metaphor version and why they were rejected.

4.3.8 Strings

Strings deserve a special mention regarding their representation, this is because while they are strictly speaking objects, their immutability means that treating them as value types is unlikely to cause problems. For this reason it was concluded that strings could have two representations in our metaphor set, depending on the preference of the user or teacher: one can either use the more accurate (but more bulky) option of treating them as objects on the heap, or they can be treated as value types that reside on the stack.

If one chooses to represent strings as objects on the heap then they would be treated as arrays of characters, with the special property of being read only after their initial declaration (due to their immutability). When they are treated as value types they don't break down: when you assign one to another you can treat them as though they are copies but not the same entity. The following simple piece of code shows how, while they are objects, their assignment to each other doesn't link them when alterations are made later. It also shows how, just like normal value types, we can't change a part of the value - if we want to make a change we have to replace the whole thing (the last line shows this especially well).

```
string x = "Hello";
string y = "world";
x = y;
y = "hello " + y; //no change to x
y[0] = 'H'; //illegal as it's read only
```

As briefly mentioned in section 4.3.6 objects on the heap that have string properties would have different representations based on the chosen of string representation: if we use the more accurate model with strings as objects then string properties would contain memory addresses, and the strings themselves would exist elsewhere on the heap as seemingly separate objects. If one uses the strings-as-value representation then string properties become simple, with the strings clearly being inside the parent object.

- Arrays
- Classes
- Constructors
- Objects

4.4 Testing and Usability

According to Spolsky [6], a valid technique for testing the usability of a program or system is what he calls "Hallway Usability Testing". <summarize what it is and why it's valid>

<explain how we used it to improve usability>

<maybe give some before and after details to demonstrate how the tests helped improve the UI and the metaphors>

4.4.1 Game levels

We took past exams and tests, and under the assumption that the questions generally try to test a focused area of understanding (for example recursion, monitoring conditions, flow of control, reference vs value type etc...) we could look at what questions students had done particularly poorly on and design levels that aim to teach that particular concept.

Another method for identifying troublesome concepts is via the literature explained above, thus we can combine the two sources and iteratively create a set of levels that targets concepts which are widely considered to be troublesome for novice programmers [7].

4.4.1.1 Elementary Levels

As explained in section 2.4 Bayman and Mayer[2] found that even simple statements could confuse students into misunderstanding code. Algorithm 4.1 shows a potential test level that uses all the constructs (which had a C# equivalent) tested by Bayman and Mayer[2]. The GOTO was kept for the sake of completeness and would not actually be present in a finished level. This simple piece of code tests multiple introductory concepts at once: variable declaration, assignment, user input, conditional statements, Boolean operation, flow of control, and the difference between variables and characters (a != 'a'). For these reasons a level based off the work of Bayman and Mayer would be reasonable as a gauge of program usability, if not end user knowledge. <We could present students with this code, after they have been taught a little, and ask them to try and trace it's flow, if they succeed even before being taught anything then it is probably too easy>

Garner et al. [8] showed that what Bayman and Mayer[2] found extended beyond just self-taught students: even formally instructed students frequently have trouble on even

Algorithm 4.1 Code that uses all the constructs tested by Bayman and Mayer[2], that had a C# equivalent.

```
int a = Convert.ToInt32(Console.ReadLine());
if (a < 2)
{

    a = 3;
    //Bad practice for a modern language:
    goto label;

}
else

    a = 4;

a = a + 6;
label:
Console.Write('a');
```

the most elementary of structures and operations. They defined the set of issues in question as "Problems with little mechanical details... Braces, brackets, semi–colons. Typos and spelling. Java and file naming conventions. Import statements (when forgotten). Formatting output. Tidiness, indenting, comments".

This sort of problem incident was found to be about 250% more common than the next most frequent non-design problem issue (design problem issues are to do with understaning tasks and formulating solutions, rather than actual code). This sort of statistic begs the question: why start teaching sophisticated concepts, while a lot of students still have trouble with loops and arrays, and in many cases, won't even learn some quite simple syntax? This question in turn should lead one to conclude that, because of the abundance of elementary misunderstandings, the suite of test and evaluation problems need to have at least one or two levels dedicated to addressing these simple concepts.

<This next paragraph might belong even more in design or implementation that the rest of these sample level explanations already do:> Taking the question of teaching syntax into consideration, it seems like a valuable (and fairly trivial) extension to include which allows users to click a button which says something like "this line has bad syntax, skip it". One might then present users with a level which is littered with alternative syntaxes for each line, while only one version of each line is valid. This sort of thing would force users to pick up on syntax errors and hopefully get them past that sort of issue.

Algorithm 4.2 A *short* example of a level with invalid syntax that the user would have to navigate around. The included comments would not be given to students.

```
int exampleVar = 2;
exampleVar = 4.0;   //exampleVar is an int not a double
exampleVar = exampleVar * 8;
int b;
exampleVar = b;     //b is assigned yet
b = 16;
b = 16 * exampleVar //no semicolon
b = examplevar;     // typo in the casing of the name
//There are many other common fundamental errors that could be included.
```

4.4.1.2 Non-Resursive Iteration

Venables et al. [9] tested a group of students by asking them to write two pieces of code which solve two separate problems, in their end-of-first-semester exam. The two programs that the students had to write for the exam were: write code that sums the numbers from 1 to N, and write code that calculates the average of a series of numbers. Of the 32 students tested only only 13 (40.6%) were "regarded as having manifested a grasp of both problems". Single student scores between questions seldom differed by more than 14%, what this shows is that the scores are probably not tied to specific questions but rather the core concept that both questions share.

Both of the questions revolve around an understanding of iteration (both using for loops and while loops), what this means is that about 60% of students tested did not have a concrete understanding of simple loop structures. Iteration is a core concept in programming, whether it is done through loops or recursion, therefore if the majority of novice programmers struggle to write or understand loops (let alone recursion) a loop-centric test level should be included with the other usability and effectiveness test levels.

Algorithms 4.3 and 4.4 show C# code that is approximately equivalent to the sample solution presented in Java by Venables et al. [9]. These 2 algorithms can be adjusted quite easily to work as test levels in our program.

4.4.1.3 Methods, Parameters, Value Types and Reference Types

As explained above Garner et al.[8] performed a study of problem incident frequency. They found that out of the 11240 problem incidents recorded during lab sessions, approximately 736 problem incidents (6.5%) were to do with method signatures, overloading,

Algorithm 4.3 Sum of N, sample solution. Converted to C# from Java code presented by Venables et al. [9]

```
int sum = 0;
for (int n = 1; n <= 20; n++)
{

    sum = sum + n;
    Console.WriteLine(n + " " + sum);

}
```

Algorithm 4.4 Calculate Average, sample solution. Converted to C# from Java code presented by Venables et al. [9]

```
int n = 0;
double sum = 0;
double x = Convert.ToDouble(Console.ReadLine());
while(x > 0)
{

    n++;
    sum = sum + x;
    x = Convert.ToDouble(Console.ReadLine());

}
if ( n > 0 )

    Console.WriteLine("Average: " + sum/n);

else

    Console.WriteLine("No list to average");
```

data flow and method header mechanics. This might seem like a relatively small number however the study covered 27 different problem areas, the most common type of problem incident, occurring 1900 times (16.9%), was to do with simple typos and syntactic misunderstandings.

The code presented in algorithm 4.5, forces the student to understand method signatures, variable scope, and value vs. reference parameters. While the content of these methods is trivial, the idea behind this sort of test level to demonstrate to the students the various differences, as well as test the usability of the method calling mechanism.

4.4.1.4 Recursive Levels

Götschi et al.[3] used two recursive algorithms to test understanding of recursion, C# equivalent versions are shown in algorithms 4.6[1] and 4.7. Respectively 48% and 35% of students tested had accurate mental models for these algorithms. This leads ones to conclude that similar algorithms are viable examples for testing and usability evaluation. As C# is not a functional language algorithm 4.6 might be too much for novice C# programmers, algorithm 4.7 was used for teaching and testing recursive mental models.

4.5 Level design and creation <NEATEN ME!>

This section discusses the various possible method that could have been used to create playable levels for our game. Broadly speaking there are two types of level creation technique: procedurally generated or hand made levels.

For creating levels we had several options: we had the options of Interpretted code(using the intermediate language), compiled code using something like CodeDom (could do final output only), a custom interpreter (disadvantage of having to cover the whole language), and pre-made levels.

Procedurally generated levels would include both levels whose sequence of events was determined by the program (but not necessarily the code behind the level) and any level the code itself is created by the computer. Programs that generate arbitary code do exist <Cite usufs stuff>, however they are only of limited value. Being able to interpret user

[1] This algorithms original language used built-in head(), tail() and concatenate() operators, these are not present in C# and thus were implemented for this example.

Algorithm 4.5 Demonstrates method signatures, passing by value, passing by reference, *and* variable scope.

```
void Method1(){

    x = 2;

}
private void Method1(int x){

    x = 4;

}
private void method1(){

    int x;
    x = 8;

}
private void method1(ref int x){

    x = 16;

}
int x;
public static void Main(string[] args){

    x = 1;
    Console.WriteLine(x);
    Method1();
    Console.WriteLine(x);
    method1();
    Console.WriteLine(x);
    Method1(x);
    Console.WriteLine(x);
    method1(ref x);
    Console.WriteLine(x);

}
```

Algorithm 4.6 A recursive algorithm that only 48% of tested students had accurate mental models for [3]. Appendix A, algorithm A.5 contains complete method definitions.

```
private List<int> Algorithm1(List<int> numlist)
{
    if (numlist.Count == 0)
        return new List<int>();
    else

        return concatenate(2 * head(numlist), Algorithm1(tail(numlist)));
}
```

Algorithm 4.7 A recursive algorithm that only 35% of tested students had accurate mental models for[3].

```
private int c(int n)
{
    if(n == 1)
        return 1;
    else
        return 4 * c(n/2) + 3
}
```

defined code could be very useful, as a user would be able to simply give the program a section of code they don't understand and they could then step though it using the metaphors. Unfortunately there are a number of difficulties associated with interpretting arbitrary C# code:

- Depending on the visualization being used, not only would the codes order have to be created, but the layout of the level itself might need to be generated: this would include generating hint and tips, and even entire maps (if we used the maze metaphor described in section X to illustrate flow of control).
- Procedural generation, if used, would *require* a custom interpreter that could understand the language relative to the API. CodeDom and reflection can perform run-time compiling of any correct code as a whole, however for this implemenetation we would need to be able to interpret code one line at a time.
- All the options other than pre-made enlarge the scope a great deal (and the scope already covers a lot ranging from cognitive psychology and teaching techniques to hardcore computer science (what the code is doing in the background and how to represent that complexity in a simple way))
- Arb-code generation would also restrict the ease of use for teaching languages other than the one it was designed for. This is because you would need to have a custom interpretter for whatever extra language you wanted to try and incorporate.

While teaching languages other than C# is not the focus of this project, many of the concepts dicussed would easlity extend to other languages such as Java. Using handmade levels would allows other languages (that follow the same paradigms) to be plugged in, in place of C# level, with basically no effort.

After settling on using pre-made levels several alterative representation and creation techniques where examined:

- A custom file format that stores all the details a level might need, but which would also require a custom 'interpreter
- An XML representation of each level - this has the benefit of having most of the work already done for you, but the downside that editing anything larger than a few commands becomes tedious and difficult as the assorted parralel lists that keep track of all the relevant information are kept far apart from one another in the

XML format - this in turn leads to the manual creation of levels being more difficult than is really neccessary. For an example of the long formatting that one has to overcome when using standard XML serialization and deserialisation look in listing 1 of appendix X

- If the standard XML serialiser were able to handle Tuples, then altering the level class to contain a single list of tuples would have made the XML output easier to work with, however because XmlSerializer requires everything to have a default constructor that takes no arguments this was not an option when using the standard serialiser. In order to investigate the viability of tuples an alternative form of serialisation was investigated: the BinaryFormatter class. While this class is able to handle lists of tuples the output is an almost indecipherable mess of text. As the output of the BinaryFormatter class is mostly in binary and therefore not all text readable an example of the output cannot be included. As the reason for investigating the serialisation of tuples in the first place was to find an easy-to-read-and-write format the output of BinaryFormatter was deemed unusable.

- After considering both alternative serialization formats it was concluded that a level editor was needed. A level editor would rectify the size and spacing issues inherent in using the original XML formatting of multiple serialised lists, and would also make interpreting the binary output when using tuples unnecessary. Therefore the decision was made to keep the original XML serialization, as even if the level editor was unavailable for some reason, one would still be able to create, read, and edit levels using a plain text editor.

Chapter 5

Design

Explain all our design decisions, why they where made, what the alternative were. Show how things progressed based on the iterative methodology explained above. The focus of the design section is not the nuts and bolts of the code but rather the overview of the program as a whole, for example whether a stack or register based system suited to design better (a hybrid was the winner). Another example of a design decision is on how to represent Console output

5.1 Functionality over aesthetics

The primary goal of this project is to create a set of unified high-fidelity methaphors that can be used in various environment. With this in mind the most important part of the metaphors is not whether they look good in this particular implementation (computer scientist != artist), but rather that they can be implemented in an assortment of different ways (real-world, images in textbooks, XNA game, or WPF form). For this reason the focus of the game is on the functionality of the metaphors that the users are presented with rather than the aesthetics of the game. For example it is more important that the user has a stream-lined understandable experience when assigning to a variable, rather than one which looks good but risks causing distractions or misunderstandings.

5.2 Representing Console IO

Several methods of representing the console were possible, including: using an actual separate black console, using a text-box on the form which keeps track of where we are in code, or making an in-game sprite dedicated to console output. We settled on the textbox as it represents what you would see in visual studio most closely, while also being explicitly a level-output console area (thus separating out debugging messages), and finally avoiding cluttering up the game window.

How the user sends strings (or other value types) to the console is still up for discussion: we could make a special method call using the method mechanism that would know to not step into the write() method, but rather to display on the form. Alternatively we could make a button that writes whatever is in the players hand onto the console (if it's a string the hand should be holding a reference to the correct location on the heap). We will settle on one and give reasons here. As of 19-02-2014 the implementation is simply to call select a method using the method-mechanism and allow the process to be automated, however the passing of references has not yet been included in this mechanism.

5.3 Register or stack based architecture <NEATEN ME>

- Register or Stack based architecture (which are we using and why)
- Cite personal correspondence webmail 11 dec 2013
  - Argue which of the two architectures we should go with and why.
  - The 'hand' can take on different roles for this: it can be like a register holding a value (and never actually empty) or it can be something of a pointer that points to the top of the stack. Regardless of the choice made here, we should probably not allow the hand to ever be blank.
  - The calculator **can** also (but perhaps should not) act a bit like a resister.
  - "There are two uses for a stack. One is for frames for LIFO control flow. We definitely want this."
  - "I think if you can unify the hand with the top of the stack and don't allow an empty hand we might be better off."

- Further debate about whether this is a stack or register system was brought up by the limitations of the first method mechanism (explained above), evaluation of that problem in particular led us to believe that because this program is meant as a tool for beginners we shouldn't make understanding of either fundamental architecture a prerequisite for using this system, and instead settled on an abstraction that uses a register and stack type system.
- If we made it **purely** a stack based system, then we have to make the user see their parameters and such explicitly on the screen as part of the current frame (potentially leading to confusion.
- However if we make it a purely register based architecture then the whole visualization we have for the stack, and current frame runs the risk of being invalid. Additionally it forces the user to perform all operation at an exprememly low level (basically at an assembler level), and while this could be useful for some people, most people know bodmas well enough that an expression such as "x + y * 2 - z" would make sense once they substitute values into each variable. Doing that expression on a registry only system would need users to perform an over-long series of operations such as
  - read reg1 y
  - read reg2 2
  - mul //overwritting reg1
  - read reg2 x
  - add
  - read reg2 z
  - sub
- The calculator (even in it's simpler form), cannot conform to either of these fundamental architectures as it is an abstraction, however it is an abstration that we need (imagine asking a beginner to evaluate an expression like (1+1+1+1+1) as a series of separate operation (much like the above example...it would impede both their learning of the subject matter and their enjoyment of the program.

5.4 The method handling mechanism.

Regarding the method calling mechanism, parameter declaration, intelli-sense, automatic type casting, and function signatures:

This topic started out regarding how parameters should be declared just before a method is called (originally it was just a question of "should I declare the parameter type and name and then automatically give it the value from the players hand, or should it be declared and given a value in two separate steps?"). It became a debate over batch declaration and assignment, and what the user is expected to provide. Regarding what the user provided the questions were about should they be given an intelli-sense sort of "pick which method you mean" and "these are the signatures we have about that method, pick one"; or whether the user should look at the code and understand the signature of the method being called and then provide each variable type and name themselves. This was the first instance where the user would not be providing all the information about functionality of the code themselves, and thus seemed more like a question for the gameplay aspect rather than the design. There are valid arguments for both mechisms: for example by making the user look up the methods available, in the current code, it would encourage a greater understanding of methods, overloading and method signature all at once. However if one looks at <I think it's code dom, I'm not sure> and mechanisms like 'this.method' and 'this.variable', variables, methods and their signatures all appear to be part of the class and thus we could provide the user with all the details that the actual compiler could see and get them to make a desision about which of the available methods is the correct one. We settled on the later machanism for selecting methods as it streamlined the usability of the method calling mechanic in addition to most closely representing what visual studio would present to the programmer.

Another mechanic behind method calling is the passing of arguments, the final verdict on whether parameter assignment should take one or two steps, was that each declaration and assignment need to be separate, if only to allow users time to cast values to the required types before they are passed in. This also would help with making it clear that we need however many variable/parameter spaces marked out on the method mechanism or current frame before we can call a method. This is not to be confused with variable declaration or assignment. Additionally we decided that it needed to be made clear that for a particular method signature the parameter names and types were already set in stone, but they all needed values to be assigned to them (this was when it was decided that unassigned parameters would show content as "???").

(much debate and explanation can be made on this point, and the decision that we made for it)

5.4.1 Handling more complicated cases

After the method mechanism was implemented the first time it behaved like this: the user would name the method they are preparing to call, they would optionally declare and assign parameters, and then they would click a button that calls the method, thus creating a new stack frame. This simple method had several issues: the easiest of these issues to correct was that users could assign to any parameters in any order and even leave parameters unassigned. The rather more serious issue was that when calling code such as "method1(method2(x))" there was no way to temporarily store the result of method2(x) so that it could be used as an argument later for method1. Additionally a 'normal user' (i.e. not a functional programmer) might read this statement as "get ready to call method1, now as a parameter for method1 get ready to call method2", when using this way of thinking we essentially start needing to be able to stack method calls inside each other, and therefore we need to create an additional stack of parameters somewhere. After much debate about possible ways to achieve this, we concluded that regardless of whether do it via the hand, the calculator, or the method mechanism we would have to implement additional stacking functionality somewhere. The various stacking ways we discussed included:

- Making the method table able to contain multiple sets of parameters, with the newest sets overlapping and obscuring the older parameters/'potential frame'. Downsides included: making it look like the oldest potential frame existed in the context of the newest frame because the older frame would still be on the method table. This method would also make the method table extremely cluttered. Implementing it this way would require significant changes to be made to the underlying API as well as the more visual elements. Example code that might work with this mechanism: "f(g(x))", even "f(g(q(5)), p(6))" should work (provided the parameters for f() had already had space reserved for them). Example code that would not work: "f(g(1) + p(2))" (this wouldn't work because we would not be able to store the result of g(1) in some sort of temporary register in order to use in later in the expression after we have evaluated p(2)), "x = f(1) + f(2)" would also not work, for the same reason. This method mechanism does not conform to either stack or registry based system architectures and seems more like a kludge.

- Removing the method mechanism table entirely and simply making any parameters

that need to be declared/remembered sit as 'empty variables' in the current frame. This method would put a marker at the start of the parameters to make it clear that they belong to the frame that you call in the future. This method had the benefit of allowing users to arbitrarily nest function calls as deeply as they needed. At a fundamental level this technique is most realistic in that it almost exactly replicates a stack architecture, however while that might be nice for more advanced users beginners are not going to know about system architectures and it runs the risk of overwhelming them with technically details that they don't need to know. Finally it also has the downside of potentially distorting the distinction between variables and parameters by essentially turning parameters into unnamed variables. Example code that might work with this mechanism: "f(g(x))", even "f(g(q(5)), p(6))" should work . Example code that would not work: "x = f(1) + f(2)", for much the same reasons as the first technique... Even "f(1+3)" currently can't be handled, which is where the next potential mechanism could come in:

- Merging the calculator and the method calling mechanism, by almost making calls to nested function act like complex substitutions into an expression on the calculator. This method would require that the state of the calculator is stored inside each frame so that when we call and return from a nested method we know where we called it and where the returned value needs to be substituted. For example we might have code that looks like this: "x = method1(method2(17) + method3(9)), we would enter the whole expression into the calculator, then let the calculator know that we want to make a 'method substitution' into method2(17), we would tell the method table to prepare a call to method2, then add the parameter 17, as we call method2(17) we could save the calculators current expression ("method1(method2(17) + method3(9))") on the stack along with the rest of the current frame. When we return from method2 we would have the return value in hand, which would then take the place of "method2(17)" in the expression. We would then repeat the procedure for the remainder of the expression, if the functionality for method2 and method3 was to simply return double their parameter value, we have the expression "method1(34 + method3(9))" in the calculator. Repeating the procedure would result in the calculator having "method1(34 + 18)". As this final expression is still in the calculator we need to make the method call on the method table while leaving 34+18 in the calculator, we haven't decided how best to handle this final situation. This stacked 'super calculator' method currently seems like the best option. It appears as if it could handle arbitrarily deep and complex method nesting without breaking.

Originally we catered for what most people would consider simple method call, examples included:

- x = f(1);
- x = f(x);
- x = f(x + 1);
- x = g(x, y, z);

To begin with this seemed reasonable, and the first metaphorical method handling mechanism we conceptualized and implemented could handle these cases without any trouble. <explain the original mechanism, perhaps include some pictures><could also include the steps involved for each case in an appendix>. Problems began to arise later during the development process once we started making sample programs. While absolute beginners probably wouldn't need to handle cases much more complicated than the above examples, the limitation of the original mechanism seemed like a grievous oversight. Cases that we need to be able to handle, but could not, included:

- x = f(1) + 2;
- x = f(1) + g(2);
- x = f(g(1));
- x = f(1, g(2));
- x = f(g(1), g(2))
- And similar cases where we had variables rather than constants.

We proposed a variety of potential solutions to this issue <explain some of the solutions as described in the ideas.lyx file>, while making the calculator behave like a tree/stack machine, where we traverse the tree to solve the problem <possibly include a picture to clarify>, this method would expose the user to far more underlying details than they need to see and would probably also require that we write temporary value to some visible location (another complication that could confuse the user).

The first and most fundamental case, where we have an expression containing method calls, highlighted part of what was missing from the original model: there was no way to include the results of methods in expressions. This realization prompted us into thinking of methods in expressions as being special place holders (not quite variables), which in

turn led us to consider how we treat method calls inside expressions as something that can have a value substituted into it. Using the super calculator mechanic, we could break solve these expressions using these steps (starting with the most simple one):

- calculator expression -> f(1) + 2;
- click method substitution button with value “f(1)”
- Clicking the method substitution button changes the calculators expression to read something like “returnValue+ 2”
- The method mechanism now has ’f’ as its named method
- <This step is a little hazy as I don’t know where to temporarily store the parameter> this step might involve (for more complex cases like f(1 + 2)) a partial simplification of the expression done by the calculator
- The parameter (1) is declared and given a value
- the expression in the calculator is stored when we call
- we call and return from f with the return value in hand
- the expression in the calcutor is returned to “returnValue+ 2”
- we substitute into ’returnValue’
- The calculator now reads “1 + 2”
- The rest is easy

A more complicated case like “f(1, g(2))” could involve steps like:

- calculator expression -> f(1, g(2))
- click method substitution button with value “g(2)”
- Clicking the method substitution button changes the calculators expression to read something like “f(1, returnValue)”
- step in and out of g() to get a value to replace returnValue (like above)
- calculator expression now reads “f(1, 2)”

Algorithm 5.1 A common recursive factorial method implementation that will not work without a more complex method mechanism

```
def normalFact(n):

    if(n <= 1):
        return 1
    else:
        return normalFact(n-1) * n
```

- At this point we can treat it like a simple method call, and just use the calculator as a guide for the parameter value. OR do it like above, and treat it like another special substitution

The only disadvantage of this method that I can sees straight away is that it does not conform with 'normal' peoples way of thinking of methods <as explained in ideas.lyx>, and instead deals with the issue by solving the most nested problem/s first and working out (rather than getting ready to solve the outer problem, then getting ready to solve the inner problems. which leads to visual stack and memory requirements). However this small issue can be side-stepped or prevented entirely (depending on how you see things) by telling the user that the rules of BODMAS still apply: in the expression f(g(1+1)) we have to do the most nested brackets first (1+1), then we move on to the next set of brackets (g(2)), and finally we do things that aren't inside brackets "f(returnedValue)". The next section explains how this issue can be circumvented or corrected.

Even after considering all that has been said about the potential benefits of having a more advanced method mechanism, we can still demonstrate most novice to intermediate concepts with the simpler one. For example, algorithm 5.1 shows the most common way of writing a recursive factorial method (this code would not be executable on our simpler method mechanism) however algorithm 5.2 (which only has a very minor change to it) performs exactly the same function while still being executable when using the simpler mechanic. The only real difference that the more compatible version saves some state in a local variable, which is in fact a potential benefit as students will see more clearly how the state changes as they recurse.

Algorithm 5.2 An adjusted recursive factorial method that works with our model.

```
    def factorial(n):

        x = 1
        if(n <= 1):
            return 1;
        else:
            x = factorial(n-1);
        return x * n;
```

Chapter 6

Implementation

Explain all our implementation decisions, why they where made, what the alternative were. Show how things progressed based on the iterative methodology explained above. Things that belong here include MVC, passing action as parameters etc...

6.1 User testing and feedback

This section details how the program was presented to users, tested by them, and them improved upon based on their critiques. The original plan for a feedback mechanism was to simply give the program to anyone in the first year class who wanted to try and then automate the process of counting their mistakes and recording the results. This method was discarded before it was ever used on students (but not before a simple email-based version had been implemented), as it became clear that there where several faults with testing usability in this manner:

- Any students who didn't understand something might simply stop, thus rendering their test incomplete.
- Other students who got stuck for whatever reason might simply click around the game experimentally until something happened, thus invalidating any error counts,
- We could only count mistakes, and thus would not get any concrete feedback about what tripped up students and why.

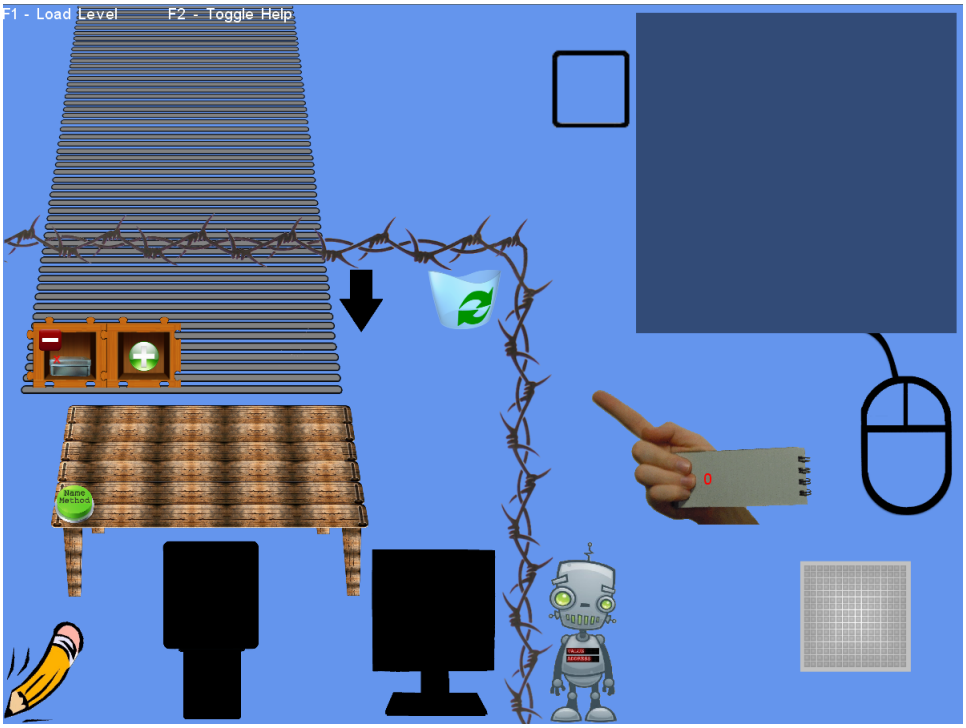

Figure 6.1: Environment prior to usability tests

Instead of testing the entire class in an automated manner, instead the initial usability tests were done on 3 people volunteers, all three test subjects had experience with programming already (one even had C# experience). The reason these people where selected was to ensure that misunderstanding of the code did not interfere with their experience of the interface. The appearance of the interface and help screen they used are shown in figures 6.1 and 6.2 respectively. This test group were not presented with anything more complicated than variables, assignment, and if statements, which they were asked to interpret using the environment, this is an abbreviated version of what they were tested using:

```
int x;
x = 9;
int y = 1;
if(x < 10)

    y = x + y;
```

All three test subject reported usability problems:

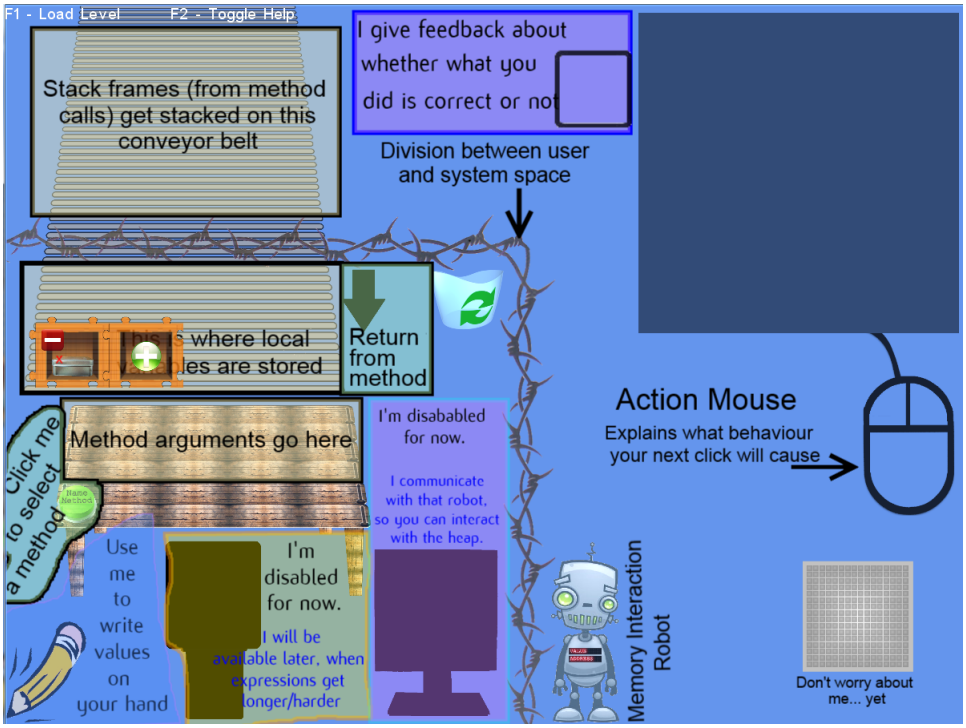

Figure 6.2: Environment with help screen prior to usability tests

- When presented with "int x = 1", they all tried to perform operations in the same order as when presented with "int x; x = 1".
- They did not find it obvious that the hand was acting as the go between for transferring values.
- The users main complaint was about what to do with a statement like "if(x < 10)".
- Two of the three users complained that the mouse buttons should be reversed for interactions with variables (Originally the left mouse button was for read, and the right mouse button was for write).
- One user noted that they found it confusing that values take their initial default value from the hand.
- One of the test subjects asked why we don't substitute into an equation for something like "x = x + y".
- A single tester reported a minor inconvenience: the hint prompt when enter a value on the hand ("<type> <value>") was inconvenient after it had been seen once.
- No testers said so explicitly, however it seemed that none of the test subjects realized there was a division between user and machine space.

- Unsurprisingly the testers also didn't know what the purpose of the game was at all, or their function in it. This was to be expected however as the only explanation they where given was a verbal summary from the tester.

Due to the the limited nature of the code they were questioned on, none of the more complicated mechanisms could be tested (such as method calling). However this was still a valid set of usability tests since almost every user had the same issues. Feedback from this set of tests led to several interface alterations:

The slightly altered interface, redesigned help overlay, and newly added introduction screen are shown in figures 6.3 and 6.4 respectively, changes of note are highlighted. <Consider not showing every re-design as it might cause clutter>

- The font of the value in-hand was enlarged to make sure it got noticed more easily, thus partially addressing the issue of users not knowing what the hand is there for.
- The mouse help sprite was enlarged to draw the users attention, its context sensitive instructions about what certain objects did was also enlarged and had its meanings clarified
- Help explaining the hand-and how it transfers values was included.
- The functions associated with left and right clicking a variable were switched.
- The whole help screen overlay was reworked, particular emphasis was placed on ensuring the user knew how important the pencil and hand-cursor are.
- Variable declaration and assignment in a single statement was altered to work the same way as declaration and assignment on separate lines.
- To be in keeping with the mechanism used for parameters, when variables are declared they start out as having unknown values (i.e. "???"). This was based on feedback saying about the default variable value coming from the hand being inaccurate and unintuitive.
- The hand-cursor has had it's movement limited so that it never leaves the user space. This was done to further clarify that there is a division between user and machine space, which must be observed.
- An introduction to the game was included that aimed to explain its purpose, this was included because subjects did not at first realise what they where meant to do, or what the purpose of the game was.

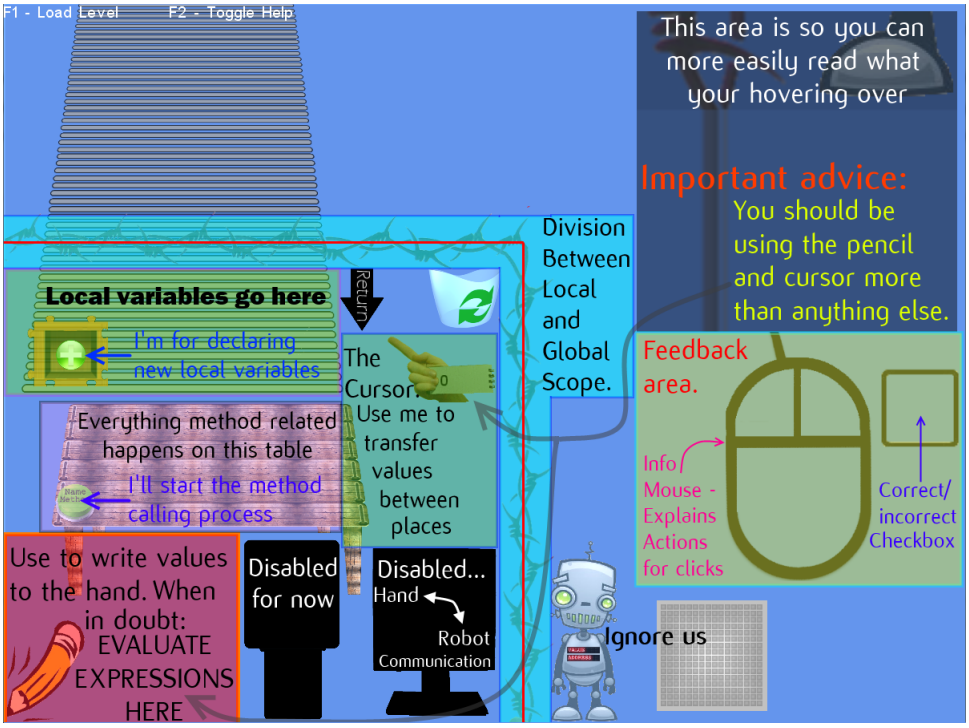

Figure 6.3: Environment with version two of the help screen.

- The hint prompt was altered so that users would only be shown the hint the first time they tried writing to the hand. This was done in the hopes that it would streamline the usability for some users, without hindering anyone's understanding.

6.1.1 Test Group 2

The next group of testers was asked to take a minute to read over the new introductory screen and help overlay, so as not to throw them in the deep end. This group was presented with similar code, except theirs also included loops and methods:

```
private int sampleMethod(int num)
{
    return num / 2;
}
private void Main(string[] args)
{
```

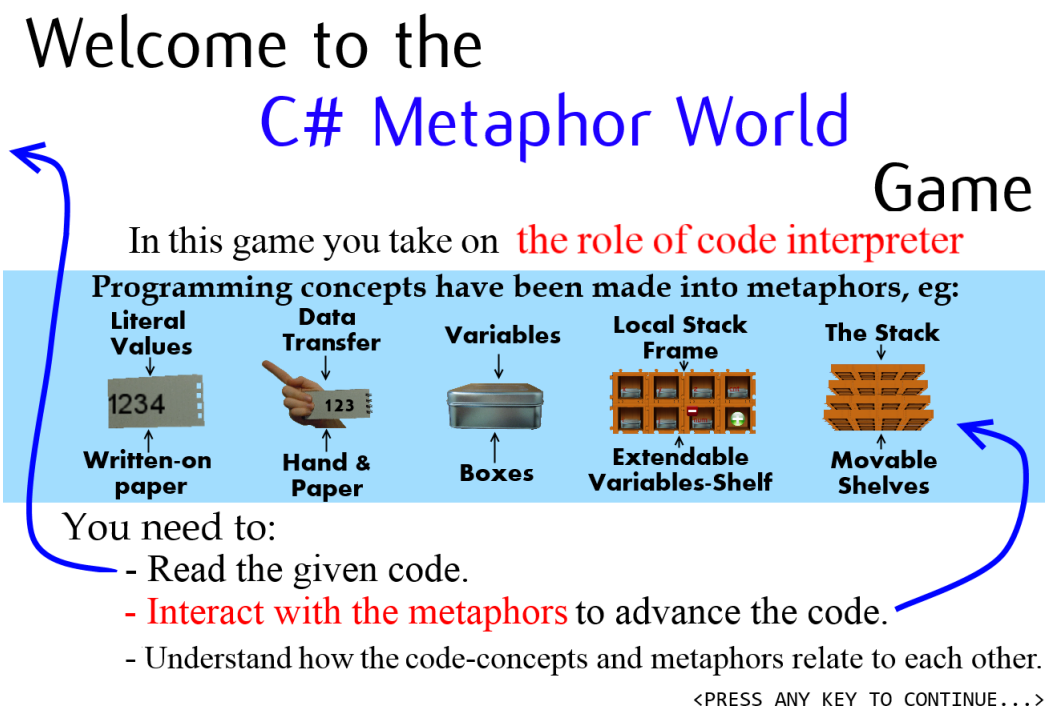

Figure 6.4: The introductory screen that was included to help inform the users as to the purpose of the game, along with their role in it.

```
        int x;
        x = 9;
        int y = 1;
        if(x &lt; 10)
            y = x + y;
        else
            y = x - y;
        while(y >= x)
            x++;
        for(int i = 10; i &gt; 9; i--)
            y = y * 2;
        y = sampleMethod(x + y);
    }
```

All three subjects had previous experience with C#, and subject two was in fact one of the first year lecturers (he had not previously been exposed to the program).

Subject one of group two had these experiences:

- The green variables area of the help was the most eye-catching for the help overlay
- Wanted to know where to get values for expressions, such as x = 1. Did not realize the function of the pencil.
- After having the pencil explained, had no trouble with "int x = 1", regarding the switch of operation orders (i.e. the new order made implicit sense).
- Had trouble figuring out what to do on the if() statement...wanted to know where to put the "true" once he had it in hand.
- No real trouble with the for() loop, did try to do the conditional first, however that is an issue more to with his understanding.
- Had no trouble with the method mechanism, either for calling or returning.

Subject two (Yusuf Motara) had a number of comments regarding the program, as well as some unusual behaviour:

- "Fix the intro screen, it has spelling and consistency errors" - consistency was to do with missing full stops.
- "The help overlay is a jumbled mess, use spacing rather than colour to highlight important areas."
- "I'd like the option to turn off the magnification area."
- "The conditional branches make no sense."
- "I feel very restricted, I'm given the illusion of freedom when there is in fact none."
- "Games only show you what you need or what you've asked for."
- Was a very experimental subject, in that he tried clicking all sorts of things erratically in order to see what did what. because the Guide interfaces look very similar, he would get them mixed up.
- Tried to enter "int x;" once.
- When presented with no format prompt for data entry (such as "<type> <value>"), the subject tried to put just a value (eg 1).

- Until it was pointed out this subject did not notice the code one the left. This probably means we need to draw attention to it somehow.

Subject three had these experiences:

- The green variables area of the help was the most eye-catching for the help overlay
- The pencil for creating initial values was not obvious.
- Had trouble with the if(). Like everyone seems to.
- When presented with no format prompt for data entry (such as "<type> <value>"), the subject tried to put just a value (eg 1).
- Had no trouble at all with the for() loop.
- Seemed fine with calling methods (program cut short as he was entering values for the parameters).

Several things to do based on this feedback:

- **Most importantly implement something to make the Boolean operations more obvious. Perhaps a conditional box that accepts bools, or 2 buttons: one which says true and one which says false. It is obvious now that help screens are not enough to make users know what to do with conditionals.**
- Note that the introductory screen, while in need of correction, did in fact help users understand the purpose of the program.
- **Re-Redesign the help interface so that when everything is there at once the important parts stand out. To do this several mock ups were created and users were asked their opinions on each one.**
- A "whats next" sort of help overlay could be implemented: it would give help specific to the current instruction. For example if the next instruction was a hand write the pencil help would be displayed, if it was a method related problem then the method mechanism help would be displayed.
- Attach the form to the side of the XNA window to make it more obvious.
- Perhaps enlarge the font of the code area.
- To prevent users from quitting by mistake, remove the Esc. key as a exit option.

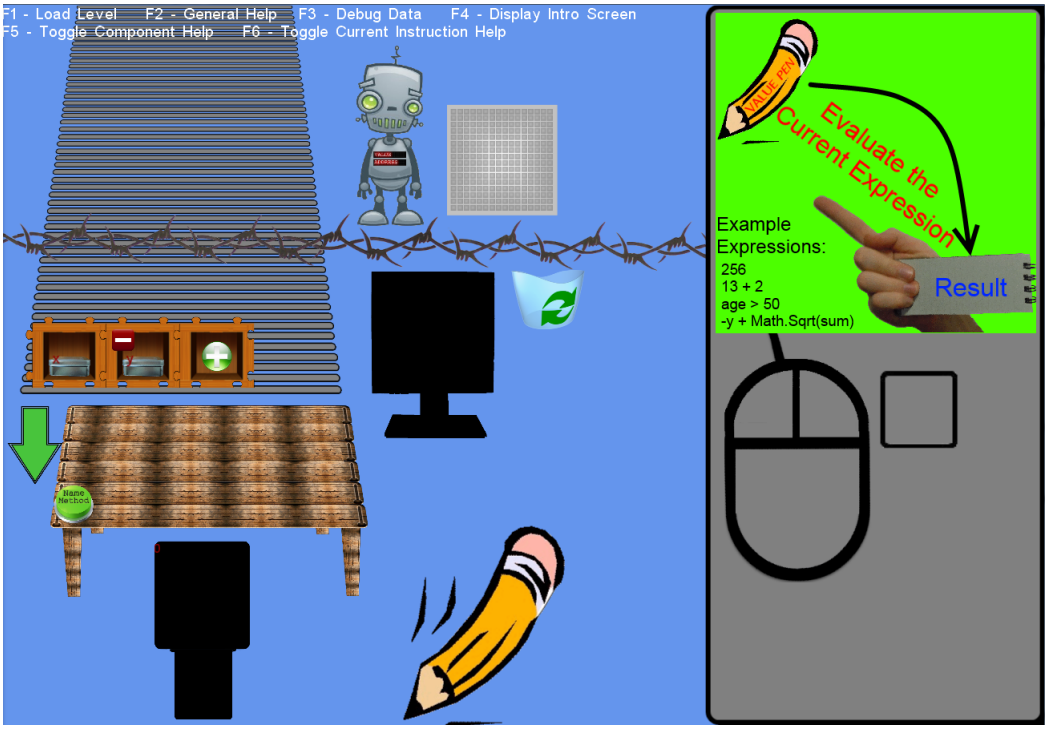

Figure 6.5:
The altered layout including a clearer division between areas.

6.1.2 Test group three

This group was presented with a re-arranged interface that made better use of space, a clearer division between user-computer-and-help space with the addition of a side panel, and contextual help. All as shown in figures X and Y.

Key ideas to take away form this group of testers:

- **The relationship between the metaphors and the actual code is not clear, Yusuf even said that the only thing driving him to interact with the metaphors was the moving of the yellow line. Basically we need to make a clearer relationship somehow...maybe through better help,** a better intro screen**, examples of appropriate operations for each metaphor, a set of help slides that explain each metaphor in more detail,** or a more gradual introduction to each metaphor on the form of 'tutorial levels' that introduce one new concept at a time.

- Students have trouble deciding which part of a line needs to be done next (biggest offender is for() loops). This can be addressed somewhat by including more sensitive code highlighting, for example int x = meth(2 + y) would be highlighted as:

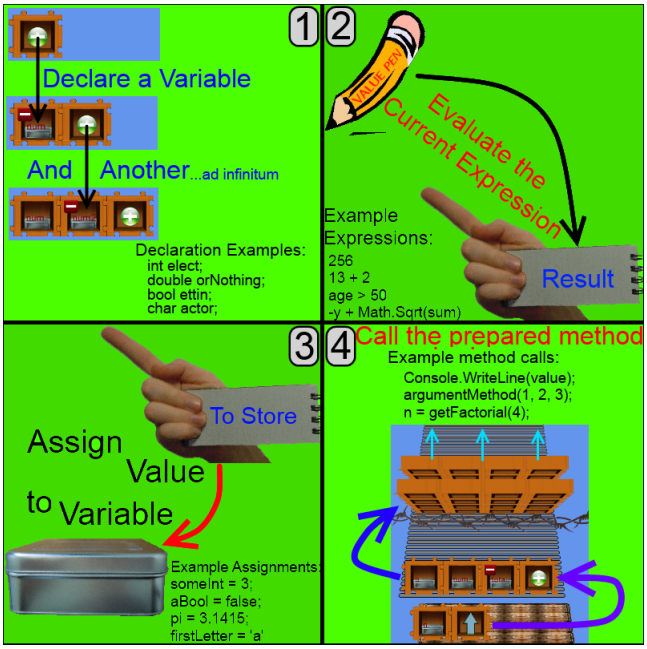

Figure 6.6:
Collection of some of the contextual help sprites. Image 1 was presented when users needed to declare a new variable, image 2 was shown whenever an expression needed evaluating, 3 was displayed whenever users had to assign to a variable, and image 4 was only shown to those who got up to using the method mechanism.

  - int x;
  - meth(
  - 2 + y
  - meth(x+y); //call
  - x = //on return

- **The context hints as they stand can be misleading,** for example "4+5" is listed as an expression, but it isn't clear that the result of said expression would be "int 9". Change help to read "4 + 5" -> "int 9".

- **Between some operations (especially assignment statements) the state of the metaphor world does not appear to change. This needs to be addressed, in part it can be corrected by making variables always display their contents.** Another fix to the apparent lack of state change is to make the values in the hand get removed when assigning them

- **Once you get into how things work it becomes fairly easy, this serves to promote the idea of using tutorial levels.**

- **Include context help for other operations (specifically method selection).**
- **The intro screen needs some clarification,** perhaps also include an arrow pointing to the code. Need to include a <press any key to continue button>. "data tranfer" needs correcting to "transfer".
- We need to remove the value from the hand once it is assigned or used as a conditional, this would make it more obvious that a hand with "true" in it doesn't have to have the value placed somewhere, and it would also be more in keeping with the metaphor of placing values in the variable boxes.
- **Consider whether more help of any kind (for example in the form of better quality, more detail, greater volume, or increased prevalence) is actually going to be beneficial. Pete said that in his experience students tend to gloss over help screen even when they are simple one. When you look at the tutorial and introductory levels of almost any game you never see users being presented with all the options that they are going to have once they are halfway through the game, for example in most RTSs players are presented with the absolute minimum number of buildings required to set up a fundamental base, and it most RPGs characters are presented with a limited number of possible abilities (with D&D being a noteworthy exception). These observations led to the realisation that students might benefit more from being presented with fewer sprites to begin with, rather than 'better' help (in whatever form).**
- **A suggestion for the displaying of variables was to to this:**
  - Unassigned variables would be visible from the side, with only their name visible
  - Assigned variables would tilted at an angle so that their name and content can be displayed simultaneously.
- if()s are still not intuitive enough, a suggestion was to make the pencil expression help examples read more like this:
  - if(**x > 10**)
  - x = **x + 1**;
  - int y = **1**;

Subject 1 was Greg again:

- Said that the contextual help did make it more clear than the first time he tested it. Both with x = 9; (the fact that 9 is an expression), and if(XXX). If() was still not smooth though.
- Thought that greying out or hiding the hand value value once it's used for something would make it more obvious that the value is no longer of the users concern.
- After getting the if() there was no trouble with the while(), for(), method calling or returning.

Subject 2 had experience with first year python

- Had trouble with almost everything...I suspect the python might have hindered, although he must be bad at CS in order to be repeating first year (a year late at that).
- Was the only person get confused by "<type> <value>" input prompt.
- even after struggling through "int x; x = 9" still had trouble on "int y = 1".
- The if() context help was not enough...he had to have it explained in painful detail.
- Found that the code highlighting was not enough to let him know what was to be done next (frequently thought that the highlighted line was the one that had just been completed)
- After eventually getting the if() there was no trouble with the while()
- "Getting used to the metaphors and what they all do was the biggest stepping stone"
- Had not covered for() in class, so everything after the while() loop was not used.

Subject 3 had no previous programming experience aside from what had been covered in class.

- No trouble with int x;
- No trouble with x = 9;

- No trouble with int y = 1;
- Had to be prompted a little with the if(XXX)
- Had some trouble with y = x + y inside the if()s body-> this is a code understanding issue I think as he wanted to try and declare y again.
- After getting the if() he had no trouble with the while()

Subject 4 did Java in school

- No trouble with int x;
- No trouble with x = 9;
- No trouble with int y = 1;
- Had to be prompted a little with the if(XXX)
- After getting the if() he had no trouble with the while()
- Had to be reminded of the order that for loops execute in, but did not have too much trouble with the for loop.
- No trouble calling the method, did have some trouble figuring out how to return with a value.
- "Getting into it is the biggest hurdle"

Subject 5 did Java in school

- Some trouble figuring out what to interact with even to start with despite the help sprites.
- After eventually getting through "int x; x = 9" did not have trouble with "int y = 1"
- Had some trouble with the if(), however after manually highlighting the relevant piece of code (x<10) he realized what had to be done.
- Had a little trouble with the while at first
- No problems with the for()

- Some trouble with the method mechanism.
- "I'm the type that jumps in and presses things to experiment and see what they do...this layout made experimenting hard and unintuitive"

6.1.3 Test group four

This group (so far) is composed of just one person, a second year with one full year of C# experience.

The only thing that caused any problem was the if() statement, after turning on the contextual help and the pointing to the example "bool true", he got it quite quickly.

Testing here was done in a slightly different manner: rather than going straight into the "all in one level", he was first asked to complete the first two lines of level 1 (int x; x =1;), this first level was using the new minimalist layout as shown in figure X. Without **any** prompting at all he completed the first two lines in under 20 seconds.

After demonstrating that the minimalist view helped people get into the simplest parts, he was presented with the methods view as shown in figure Y. He quickly completed everything up to if(XXX) at which point he needed help in figuring out what to do (not quite as much as previous people though). It may have been a mistake to use the minimalist view on just those variable lines, and we should probably have included at least one Boolean conditional when testing the minimalist view (this might have helped him get the if() even faster). Overall he was much faster than previous subjects, the only major difference was the gradual introduction of sprites rather than lumping them in a seemingly orderless world without any adjustment period.

This test prompted an alteration to the different views: instead of having three set modes, the level format was altered so that any sensible combination of sprites could be presented. For example if the goal of a level was to introduce conditionals you could include the boolean pen, local variables, and nothing else. Code such as this could be used to introduce conditionals:

```
bool x;
if(1 < 2)
    x = 3 >= 4;
```

```
else

    x = 3 <= 4;
```

Figure X shows a selection of various sprite arrangements for different situations. Additionally this phase made another issue regarding if() statements more clear, some students would work out the conditional in their heads and then try to immediately do the first line of code for the if() block. Using the aforementioned code in this section as an example, students would work out "$1 < 2$" and despite the if() statement being highlighted they would immediately go on to try and write the result of "$3 >= 4$" on their notepad. A possible work around for this issue would be to explain in the intro screen that the highlighted section of code is what *must* be done next.

6.1.4 Test group five

By this stage of testing there were not many potential participants left who had not already tried the game. The only 'test' subject for this round was a in fact a lecturer, nevertheless several good points where brought up:

The first qualm was about how the barbed wire barrier was unnecessarily intimidating, and gave the impression that what happens in memory space is something to fear and avoid at all costs. Several alternatives were suggested, until we settled on a 'caution-tape' barrier with a custom message; this was an unintimidating was to show memory space separation while also informing the user of the reason for the barrier (as shown in figure X).

The second suggestion was even more valuable than the first: having multiple pencils was fine, however the arrangement and behaviour at the time of testing was unintuitive. For example there were no significant transitions between clicking on a pencil, writing down a value, and then actually holding said value. Secondly it is somewhat disconcerting that you somehow write on the piece of paper it your hand with the pencil that isn't being picked up in any way. This might seem like an almost petty complaint, however if you consider that metaphors being presented in an unnatural manner could easily throw off a novice, this sort of alteration becomes fairly significant.

To rectify this issue several changes were made: the multiple pencils were turned into multiple notepads with their own individual pencils, when the cursor-hand was altered so

that a value-paper was only shown when the user was actually holding a value, and finally when writing a value on one of the notepads the cursor holds the pencil that belonged to the notepad. The different states that the hand, notepads, and pencils go through are illustrated in figure X.

6.2 Validation of metaphors

After polishing the metaphor set through usability tests (as described in section X) up to a point where further improvement becomes incredibly difficult, the next phase of development is to seek verification of the validity, fidelity, and accuracy of the metaphors so far. Quantitative validation would be ideal, and could be done by taking a group of novice programmers, splitting them into two groups, giving only one of the two groups access to the metaphors and associated game, and then comparing the final marks of both groups against each other. However this sort test could be considered unethical as if the metaphors turn out to help (or somehow even hinder) the group that was exposed to them, then the other group could argue that they were put at an unfair disadvantage. Furthermore this kind of test, even if done in an ethical manner such as through voluntary participation, would take up to three months and require a fresh group of novice programmers - thus it is also infeasible due to time constraints.

An alternative method to criticise and evaluate the metaphors and the game, would be to take a more qualitative approach: for example a survey detailing the metaphors, demonstrating how they work, and then asking various questions of the participants. This sort of test could be presented to both novice and more experienced programmers. However as the primary goal would be to evaluate the quality of the metaphors, feedback from more experienced programmers would be far more valuable, as they are more likely to already have high-fidelity mental models of programming which they can use for comparative purposes.

Questions could include:

1. If you had to explain concept X as a real-world metaphor, how would you do it?
2. Can you think of any way your mental models of particular concepts do not interact well with each other?
3. Does the metaphor we give for X compare with how you imagine it working?

4. Can you find fault with metaphor X, if so what would it be?
5. If you could alter one thing about these metaphors what would it be?
6. If you were presented with this piece of code: XXX. which series of metaphor interactions do you think corresponds to what actually happens <provide a series of screen shots>

The question regarding how a particular piece of code would be represented using the metaphors, could be altered and enhanced to a point where it could be given to novices: explain the metaphors, give some choice examples of code snippets (for example an assignment, and an if()) and accompanying **correct** metaphor use, and then ask them to choose certain metaphor/code flow sequences. The questions could be varied:

1. Given this code, which of these sequences of metaphors is correct?
2. Given this code, what order do the lines execute in? (This would be to gauge their actual understanding of code flow for comparative purposes).
3. Match the section of highlighted code on the left to the corresponding metaphor-action on the right.
4. These X numbered pictures are a shuffled version of the correct sequence of actions (for our metaphors), please arrange them so that they are in the correct order.

The second type of survey could be given to novices as well as more experienced programmers. Section X <the usability section> demonstrates that the metaphors can be made into a playable game, this survey would serve the dual purpose of testing quality and demonstrating whether the metaphor set accomplishes one of its other basic goals: **being usable it a text book.**

6.2.1 Incomplete Survey:

<Include a description of the metaphors as they stand...ie no history or enhancements, only what they are right now>

get sum of numbers divisible by of 2 or 3

```
int sum = 0;
int counter = 6;
while(counter >= 0)
{
    if(counter % 2 == 0 || counter % 3 ==0)
        sum += counter;
    counter--;
}
Correct sequence ->
    declare sum
    get 0
    assign sum
    declare counter
    get 6
    assign counter
    get true
    consumeBool
    get true
    consumeBool
    get 6
    assign sum
    get 5
```

find largest common factor

factorial

prime number code

largest element in an array <can't do it without including objects>

6.3 Requirements of the hand and a calculator

In the above section I discussed how there is currently no mechanism for handling method calls that are part of expressions, as the values are lost before they are assigned to anything. After discussing the issue we concluded that this sort of situation is actually not all that important, for a number of reasons:

- Single line expressions that contain method calls are above a beginner level
- If you do need to have the results of method calls used in expressions there is no reason not to simply break up the complicated expression into several shorter simpler lines, for example x = f(g(1)) + g(3) + 4 can be broken down into:
  - temp = g(1)
  - temp = f(temp)
  - temp2 = g(3)
  - x = temp + temp2 + 4
- Additionally breaking long expressions down into sub-expressions is closer to what being done in the background by the compiler (you just don't see the temporary variables, or the placement of temporary values on the stack)
- The main focus of this project is not to make the users able to understand and execute long complicated expressions, but is in fact primarily to make them understand more fundamental ideas such as: control flow, call and return, variable declaration and assignment, arguments and parameters etc...

We can split the games concepts or interactions into 2 areas, those that require the user interact with the in-game calculator and those which can be done either without any calculator at all or using a real-world calculator:

No calculator required	Need in-game calculator
control flow	Automated BODMAS
frames	method calls **in** expressions
arguments	<think of more>
call	
return	
assignment	
reference types	
simple variable reads	
<think of more>	

This table highlights how we can almost entirely do without the in-game calculator, and perhaps for certain situations this might be better for students (such as when they are learning simple expressions it isn't necessary to make them look up and substitute for

every step of an equation, if they have a calcuator in their hands that can do it for them). What this leads to is a question about whether having both the hand and the calculator is strictly necessary, it can be argued that we don't really need the hand if expression in the calculator always simplify down to single terms regardless and that the calculator could therefore take the place of the hand entirely (by making it take on the role of the hand in addition to what it already does). The downside to excluding the hand in favour of the calculator is that when we have an expression like "x = x + y + z", there is no intermediary available to store each variable value before we substitute it into the calculator. One can argue that this is not a necessary step as you could break it down into steps such as "substitute 'something' into x" and "select where to read 'something' from", however I feel that this is an unintuitive order to do things in. Additionally there is a major benefit to always having the hand as your intermediary value storage device:

The hand acts as a means of communication between **everything** in the user space, if anything needs to communicate with anything else it needs to go through the hand. This essentially makes anything the hand interacts with seems more like a service than a. Put a slightly different way we can say that the hand is the users means of interacting with all the manipulatives, and that all behavior needs to go through the user. This everything-through-the-hand behaviour also allows for a consistency of communication: whether we are reading from or writing to the robot, the calculator, a variable, a parameter, or anything else, the data being send goes through the hand (essentially turning the hand into the sole communication device).

As explained in the above table, a lot of time we can get away without using the in game calculator at all. Therefore, some of the time, we can completely exclude the in-game calculator, and simply make the user look up values and work out expressions using their real-world calculators. The advantage to doing puzzles/games this way is that the user doesn't have to get bogged down with explicit reading and substitution of variables, and communication with the calculator, when learning the fundamental concepts and can instead focus on whatever the current puzzle/question is trying to teach (method calling, arguments, reference types etc...).

6.4 MVC

6.5 Extensibility

In this section we will explain how the games design allows for high extensibility, by detailing the steps involved in implementing a Boolean value consumption mechanism. During the testing phase, described in section X, the need for such a mechanism presented itself fairly early on; however such a mechanism ran the risk of confusing students and adding unnecessary complexity. For this reason the idea was taken under advisement until almost the end of the test phase, at which point it became clear that such a mechanism was in fact necessary.

The implementation of this mechanism is a perfect example of how extensible the system is, because it introduces new visual components, new user interactions, and changes the way things work at a control level.

6.5.1 Creating the sprite

The first step is to add the sprite to the interactiveSpriteCollection object, it does not yet have any interactive functionality:

```
interactiveSpriteCollection.Add("booleater", new dynamicSprite...);
```

At the start of every level the sprites that are enabled are determined by the level in question, unless we include this new sprite into one of the modes it will not be enabled and thus will never draw or update. This also prevents the sprite from doing anything if we are in a mode that does not allow for conditional operations.

```
if (mode.Contains("conditional")
    enabledKeys.Add("conditionals");
```

Now the sprite will draw to the screen, however it has no behaviour associated with it yet. There are two behaviours that need to be exhibited by this sprite: it needs to only get drawn when the current levels next operation is a conditional, and when clicked it needs to send a message to the world tracker (the control class) saying that it has recieved an interaction request. Before altering the control class we will set up a some dummy delegates just to ensure that the behaviour is correct, this enhancement is to hide and show the sprite and is back were we originally add the new sprite to our sprite collection:

```
Predicate<string> opIsGiveBooleanConditional =
    delegate(string key)
    {
        return DateTime.Now.Second % 2 == 0;
        //return theWorldTracker.isNextOperation("consumeBool");
    };
interactiveSpriteCollection.Add("booleater", new dynamicSprite...
, operationIsBooleanConditional);
```

The above code results in the sprite hiding itself every other second, the commented line is what we will use later one the control class actually knows what "consumeBool". The next place holder delegate will eventually send messages to the world tracker, like the previous delegate the actual message line is disabled as it would send an unrecognised message:

```
Action consumeBool =
    delegate()
    {
        Console.Write("clicks are working");
        //theWorldTracker.performOperation("consumeBool", freePlay)};
    };
interactiveSpriteCollection.Add("booleater"...
new interactiveRegion(... consumeBoolean...)...
, operationIsBooleanConditional);
```

If this sprite did not require an alteration to the more fundamental behaviour of the system the above enhancements would be enough to do a great deal already. However as the inclusion of this sprite alters the way the needs to behave, alterations need to be made in some other areas. The first step is to make the controller (the world tracker) recognise "consumeBool" as a legal operation:

```
case("consumeBool"):

    Console.WriteLine("No behaviour for this op yet");
    //succeeded = consumeBoolFromHand(finishingAction);
    break;
```

The above code fits in the switch block of the controllers performOperation method, which attempts to perform a given operation and return a boolean based on whether the operation was successful or not. The code does not have a definition for consumeBoolFromHand yet, and there are two ways we can do it: either using delegates or a normal method declaration. The important part of the controllers code for consumeBoolFromHand looks like this:

```
if (thePlayer.examineHeld().readType() != "bool")

    return false;

thePlayer.removeHeld();
finishingAction();
```

The above code is surrounded by some standard exception handling. By this stage of alterations the new sprite is almost complete, all that has to be done is remove the appropriate place-holder lines and uncomment the actual checks. Now when we click on it with a boolean in-hand it will consume the boolean and advance the current level to the next operation, and it will only display when the next operation is one to consume booleans.

This enhancement took under 20 lines of actual code to spread over three areas, the only things left to do in order to make this enhancement fully integrated into the system is to include the new operation into the levels and level editor.

Chapter 7

Results

Describe what we ended up with - for metaphors made, the level of success with students, observations about this sort of teaching method.

Chapter 8

Analysis of Results

Explain the results and why we think they came out as they did. Additionally explain what the implications of the results are.

Chapter 9

Future Work

There are a large number of ways in which this work could be altered, enhanced, or extended in the future. From the perspective of ease of use the most obvious extension would be to implement a code interpreter (as described above in section X). An extension of this nature would allow not only exereinced users to create levels, but would also allow nivices to step through code which they would like o understand using the program. This extension would be a relatively easy one once C# 5.0 is finished whose design specification include Compiler as as Service (https://en.wikipedia.org/wiki/Microsoft_Roslyn) <The discussion about why this was not done belongs in the implementation and/or design section/s>.

With the current generic nature of the metaphors and environment there is no real call for any sort of procedural map/level generation, but perhaps procedural generation of hints and tips based on the line being 'read' could be included.

If neither of the above enhancements are feasible, a fast easy to use level creator wouldn't go awry <I might actually make one>.

Another future enhancement (or even separate project), could be to include the mazes and maps that I proposed to begin with, the focus of such a project would be on teaching flow of control over flow of data (as explained in a previous section). This sort of alteration would benefit greatly procedural map generation.

Different implementation of the same set of metaphors and manipulatives could add credibility to the metaohirs, for example a 3D version of the metaphors and general game, or one that works online.

We devised a number of potential metaphors and manipulatives (and each one had various possible implementation methods), thus an enhancement that allows different metaphors/ sets of metaphors to be selected would allow for easy testing of metaphor validity and fidelity as well as allowing a change of perspective. The MVC design of the manipulatives API is highly conducive to this kind of enhancement.

Finally one could give the metaphor API to some intermediate game development students and ask them to implement a simple version of the game: this would serve only to prove the simplicity and ease of use of the API.

Chapter 10

Conclusion

We do not have any conclusions yet.

Appendix A

Exam Questions and Corresponding Level Code

A.1 Method Examples

A.1.1 Method Signatures

If I have a method called Method1 that does not return a value and does not take any parameters, what would its signature be?

(a) private int Method1()

(b) private void Method1()

(c) private bool Method1()

(d) private string Method1()

(e) none of the above

Corresponds to code such as:

Algorithm A.1

```
void Method1()
{}
private void method1()
{}
private void Method1(string x)
{}
Method1()
```

Algorithm A.2

```
class X
{
public static int x;
public int y;
public X (int a, int b)
{

    x = a;
    y = b;

} // constructor
} // class X
...
X var1 = new X(3, 4);
X var2 = new X(7, 8);
Console.WriteLine(String.Format("{0}, {1}", X.x, var1.y));
```

Algorithm A.3

```
private void m1(string sx)
{

    sx += " World!";

}
string sw = "Hello";
m1(sw);
string result = sw;
```

Algorithm A.4

```
    private void m2(List<int> xs)
    {
        List<int> result = new List<int>();
        for (int i = 0; i < xs.Count; i++)
        {
            result.Add(xs[i] * 2);
        }
        xs = result;
    }
        ...
    List<int> ws = new List<int>() {7, 11, 12};
    m2(ws);
    int b = ws[1];
```

Algorithm A.5 The complete code for algorithm 4.6.

```
private int head(List<int> list)
{
    return list[0];
}
private List<int> tail(List<int> list)
{
    List<int> temp = new List<int>(list);
    temp.RemoveAt(0);
    return temp;
}
private List<int> concatenate(int head, List<int> tail)
{
    tail.Insert(0, head);
    return tail;
}
private List<int> recursiveExample(List<int> numlist)
{
    if (numlist.Count == 0)
        return new List<int>();
    else

        return concatenate(2 * head(numlist), recursiveExample(tail(numlist)));
}
```

Listing 1 Example of the standard XML serialisation of a level, before including selective highlighting.

<?xml version="1.0" encoding="utf-8"?> <Level xmlns:xsi="http://www.w3.org/2001/XMLSchem instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Introduction to Methods" description="This is a level there to try and test everyhting in one go...at least the simpler everthings :)"> <availableMethodSignatures> <string>Console.WriteLine(char valueToWrite)</string> <string>sampleMethod(int num)</string> </availableMethodSignatures> <code>private int sampleMethod(int num) { return num / 2; } int x; x = 9; int y = 1;

```
if(x &lt; 10)  y = x + y; else  y = x - y;
while(y >= x)  x++;
for(int i = 10; i &gt; 9; i--)  y = y * 2;
y = sampleMethod(x + y); </code>
<lineOrder>
    <int>6</int> <!-- declarevariable int x -->
<int>7</int> <!--directhandwrite int 9 -->
    <int>7</int> <!-- assign x-->
<int>8</int> <!-- declarevariable int y -->
    <int>8</int> <!--directhandwrite int 1 -->
    <int>8</int> <!-- assign y-->
<int>10</int> <!--directhandwrite bool true  -->
<int>11</int> <!--directhandwrite int 10 -->
    <int>11</int> <!-- assign y-->
<int>15</int> <!--directhandwrite bool true  -->
<int>16</int> <!--directhandwrite int 10  -->
    <int>16</int> <!-- assign x-->
<int>15</int> <!--directhandwrite bool true  -->
<int>16</int> <!--directhandwrite int 11  -->
    <int>16</int> <!-- assign x-->
<int>15</int> <!--directhandwrite bool false  -->
<int>18</int> <!-- declarevariable int i -->
    <int>18</int> <!--directhandwrite int 10 -->
    <int>18</int> <!-- assign i-->
    <int>18</int> <!--directhandwrite bool true  -->
<int>19</int> <!--directhandwrite int 20 -->
    <int>19</int> <!-- assign y -->
<int>18</int> <!--directhandwrite int 9 -->
    <int>18</int> <!-- assign i-->  <int>18</int> <!--directhandwrite bool fals
<int>21</int> <!-- preparemethod sampleMethod(int num) -->
    <int>21</int> <!-- directhandwrite int 31 -->
    <int>21</int> <!-- assignparameter num -->
    <int>21</int> <!-- callfunction -->
<int>3</int> <!--directhandwrite int 15 -->
<int>3</int> <!--return -->
<int>21</int> <!-- assign y-->
</lineOrder>
<completeLevelInstructions>
<string>declarevariable int x</string> <!-- 6 -->
<string>directhandwrite int 9</string> <!-- 7-->
<string>assignvalue x</string> <!-- 7-->
<string>declarevariable int y</string> <!-- 8-->
```

Bibliography

[1] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, March 2007.

[2] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM*, 26(9):677–679, September 1983.

[3] Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. *SIGCSE Bull.*, 35(1):346–350, January 2003.

[4] Janet Rountree and Nathan Rountree. Issues regarding threshold concepts in computer science. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 139–146, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[5] Allan Collins. Toward a design science of education. technicalreport no. 1. Technical Report 1, Center for Technology in Education, New York, NY., http://www.eric.ed.gov/PDFS/ED326179.pdf, January 1990.

[6] Joel Spolsky. The joel test: 12 steps to better code. Webpage - http://www.joelonsoftware.com/articles/fog0000000043.html, August 2000.

[7] Peter Wentworth. Private Correspondence, February 2014. Email correnspondence between author and Peter Wentworth.

[8] Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05, pages 173–180, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[9] Anne Venables, Grace Tan, and Raymond Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 117–128, New York, NY, USA, 2009. ACM.