Programming Osmosis: Knowledge Transfer from Imperative to Visual Programming Environments

Dale Parsons
School of Information Technology
Otago Polytechnic
Dunedin, New Zealand
dale@tekotago.ac.nz

Patricia Haden
School of Information Technology
Otago Polytechnic
Dunedin, New Zealand
phaden@tekotago.ac.nz

Abstract

Computer science educators continue to develop new ways to support the teaching of introductory programming. Among some of the most popular new tools are Visual Programming Languages (VPL), which provide graphical interfaces for code construction and program display. In this paper we explore the use of Alice, a sophisticated VPL for building 3D animated scenes. When used in conjunction with a traditional programming language, we found minimal transfer from the imperative context to the visual environment. We also found that students struggled to make the connection between work in Alice, and "real programming". We suggest caution when using curricular materials which, while technologically appealing, may not be pedagogically appropriate.

Keywords: Programming education, Visual programming languages.

1 Introduction

For nearly three decades, educators have wrestled with the teaching of computer programming. Currently, they are still exploring and debating different languages (e.g. Duke, et. al., 2000; Giangrande, 2007), programming environments (e.g. Kelleher & Pausch, 2005), programming paradigms (e.g. Brilliant & Wiseman, 1996) and curriculum content (e.g. Howell, 2003). Unfortunately, recent large studies (e.g. McCracken et. al., 2001) have shown that, across methodologies and across institutions, students continue to fail to achieve the desired level of competence in early programming courses. This failure is puzzling, since computer programming seems, inherently, no more intellectually challenging than maths or physics of any number of other subjects which are more successfully taught.

This quality assured paper appeared at the *20th Annual Conference of the National Advisory Committee on Computing Qualifications* (NACCQ 2007), Nelson, New Zealand. Samuel Mann and Noel Bridgeman (Eds).

Some explanation of this phenomenon can be found in the large body of literature that looks at the acquisition of programming skill from a cognitive perspective (rather than a pedagogical one). The study of the psychology of programming has identified a basic mismatch between the way humans think and the way computers "think" (Hoc et. al. 1990). The novice programmer brings in well-developed cognitive models of general problem-solving acquired through years of experience with logical manipulation. These cognitive models are not based on the "sequence-branch-loop" protocols of computer programs. What has always worked suddenly does not. Novices must effectively relearn problem solving to become expert programmers. Their existing knowledge is in conflict with the knowledge they are trying to acquire, making the acquisition of programming skill arduous and often frustrating.

If it is so difficult for students to learn to "think like a computer", one could presumably facilitate programming education by making the computer think more like the student. Or, at least, appear to think more like the student. That is, one could allow novice programmers to experience problems and express solutions in an environment that more closely matches their existing cognitive model of problem solving.

This philosophy is most obviously expressed in the development of high-level programming languages that allow the programmer to describe an algorithm in "English-like" statements, rather than in assembler or binary. The introduction of high level languages in the 1970s made programming much more generally accessible.

More recently, advances in graphical capabilities have allowed the development of "visual programming environments" (e.g. Edwards, 1988; Ichikawa & Hirakawa, 1987). In these contexts, flow of control and program state are represented visually, often with animation (see section 3 below for a more detailed discussion). Programmers no longer need to mentally translate the computer's real-time behaviour into a linear series of statements. Rather they effectively construct a picture or diagram of the computer's behaviour, which allows a problem solution to be expressed in terms of the direct manipulation of the computer's state. This

approach is held to be more closely aligned with the novice programmers' existing mental model of problem solving (Materson & Meyer, 2001). Early explorations of teaching with visual programming environments have been encouraging (e.g. McKeown, 2004). While nothing is going to make learning to program an easy task, it does appear that, for some students, visual programming environments allow the expression of problem solutions in a natural way.

In the present study, we have focussed on the role a visual programming environment might play in the learning of a specific aspect of programming semantics – simple flow of control. The logic of branching (if statements and other conditionals) and looping (while, for, repeat, etc.) is a consistent thorn in the side of programming students and educators. Our own experience is that students often fail to recognise the situations where looping and branching are required, and struggle to see what these constructs actually do. Standard flow of control patterns do not seem to match well to our students' *a priori* problem-solving templates. One of the oft-cited strengths of visual programming environments is that they represent flow of control naturally (Dann, Cooper & Pausch, 2000). For example, students can understand the purpose of looping when they can see some action happening repeatedly.

If it is in fact natural to *comprehend* the role of flow of control structures in a visual programming environment, might it also be natural to *use* flow of control constructs in such an environment? Specifically, to what extent will students spontaneously recognise the role of branching and looping in an environment where the use of such a construct is translated directly into observable behaviour? In this study, we gave students unstructured access to a visual programming environment while simultaneously teaching them flow of control in the traditional imperative paradigm. We then observed the extent to which the students incorporated structured flow of control into their visual programming. Our assumption was that if flow of control is truly naturally represented in a visual environment, students would transfer their imperative knowledge to the visual environment without explicit direction.

In section 2, we will briefly review the psychology of programming literature. In section 3, we will discuss visual programming environments in general. In section 4, we will introduce Alice, the tool used in this study. In section 5, we will present the results of our exploration. In section 6 we will discuss our conclusions regarding the use of visual programming tools in introductory programming courses.

2 The Psychology of Programming

Cognitive psychologists have been interested in computer programming for essentially as long as the activity has existed. Seminal work by Brooks (1978), Jeffries (1981), Pennington (1987), Shneiderman (1979) and Weinberg (1971), among others, connected the mental discipline of computer programming to the larger literature on human problem-solving.

In the effort to understand how programming is done, much of the focus in the literature has naturally been on the difference between those who can program (generally termed "novices") and those who cannot (generally termed "experts"). Various authors (e.g. Adelson et. al., 1981; Davies, 1990; Koenemann, 1991) have observed that becoming an expert programmer is not simply a matter of learning the syntax of a programming language. Rather the expert has developed a problem-solving strategy that is qualitatively different to that of the novice. The novice can only visualise a problem solution the way human would do it; the expert can see the solution as the computer would do it.

Fix, Widenbeck, & Scholtz. (1993) identify several stages on the journey from novice to expert. At each stage, the developing programmer has amassed a larger set of patterns or templates for "thinking like the computer". When faced with a new programming problem, the novice starts with a conceptually blank page, and must cast about blindly for a solution, often relying on trial and error. The expert, in contrast, can often recognise the novel problem as an exemplar of a class with which he is familiar, and for which he has an existing solution template. Thus the process of becoming a good programmer is essentially that of building the necessary templates for problem classification and solution. These templates comprise the expert's *cognitive model* of computer programming.

Traditionally, programming educators have driven the development of these cognitive templates through repetition. Programming textbooks often have, for example, a chapter on if-statements containing examples of their use, followed by dozens of exercises requiring them. In this way, we attempt to teach the student how to "solve problems the way the computer does". As discussed above, this approach has met with only limited success. Thus educators are now trying to create environments where the gap between human and computer is not so large, where students can use their existing problem-solving skills more effectively. In this way, the programming templates they must acquire can be built upon existing cognitive foundations.

3 Visual Programming Environments

The use of visual stimuli in education is, of course, not new. Scale models, visual demonstrations and textbook illustrations have been used in all fields to facilitate learning. Some things are simply easier to understand when one sees them than when one reads about them. The use of visual representations in computer programming was for many years hampered by the restricted output options for computing machines. Paper printouts and command-line interfaces severely limit the opportunities for effective visual display, but recent advances in technology have allowed for great strides in this area.

Programming tools that rely heavily on visual display fall into two general categories: those where the programming language is itself largely graphical, and those where the output or result of a program is a visual

display, usually an animation of some kind. Some environments combine the two styles.

Among the first uses of visual representations in computer programming were algorithm animation systems (e.g. Brown & Sedgewick, 1984). Complex computer algorithms often require manipulation of data that is difficult to understand when described verbally, but easy to follow if one can "see" it happen (consider, for example, the "bubbling up" of a large number in a bubble sort, or the repeated combinations of sub-trees in a Huffman encoding).

Another important use of graphical representation in programming is the Visual Programming Language (VPL) (see, for example, Edwards (1988) and Wester, Sint & Kluit, 1997). VPL systems can be thought of as essentially dynamic flowcharts. Programs are written by combining functional units that describe the flow of data and control in the program. VPLs eliminate much of the burden of syntax (e.g. the specific location of brackets and semi-colons is rarely relevant). VPLs have been shown to be much easier for a novice programmer to use than the traditional statement-based programming language. In fact, this style of programming has proved so accessible that it is used in a popular children's toy, the LEGO Mindstorms Robot (www.mindstorms.com). With the Mindstorms system, users can program a small motorised vehicle by dragging and dropping icons to build up a sequence of instructions, including branches and loops. Mindstorms is marketed as suitable for children ages 10 and older. Several computer science educators have used Mindstorms successfully at the secondary level to teach introductory computing (Hood & Hood, 2005; Goldman, Eguchi, & Sklar, 2004)

Several modern visual programming environments support complex computer animation systems. In these environments, the programming "language" represents a series of actions in a screen animation. Programmed commands translate directly into observable actions in the animation. These systems include the venerable Karel the Robot (Pattis, 1994), PigWorld (Lister, 2004) and Raptor (Carlisle et. al. 2005).

The strength of visual systems as teaching tools is that they represent a program's internal state and flow of execution in a way that seems natural to novice programmers. Students who are puzzled by the role of primitive variables or iterations can clearly understand the notion of pigs on the screen or a robot making a series of left turns.

4 Alice

The programming environment used in the current study is Alice (Cooper, Dann and Pausch, 2000). Alice combines both categories of visual programming described above – the language is highly visual, and program statements translate directly into screen animation. Alice is considerably more elaborate than the other environments of this type however, as it comprises a 3D-graphics engine and a large library of animated 3D models.

Alice was developed by Carnegie Mellon University, and is available for free download at www.alice.org/downloads/authoringtool. Alice has been used in several introductory programming courses at both secondary and tertiary levels, with promising results (e.g. Adams 2007).

An Alice program is essentially a movie script for the characters on the Alice screen. Alice characters have a set of pre-determined parameterised commands (e.g. Move(n) and Turn(n)) that control their behaviour. Users can create new named constructs (effectively subroutines) by combining existing commands. The state of characters (e.g. their location) can be inspected and modified at run-time. An example Alice screenshot during programming is shown in Figure 1.

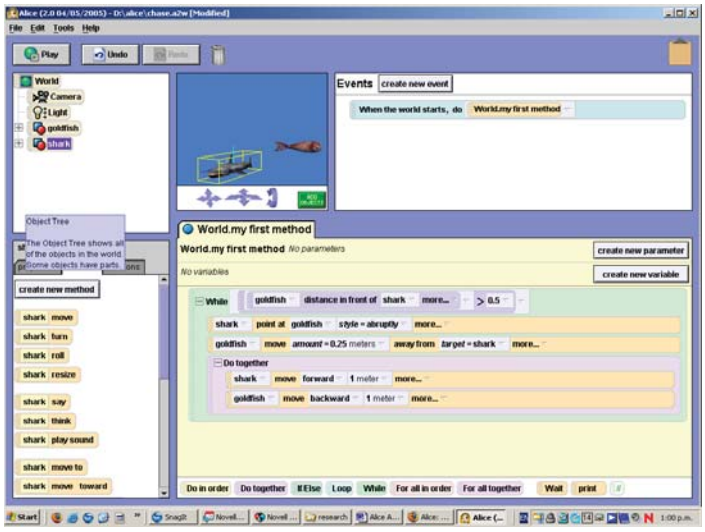

Figure 1. Alice Programming Environment

Alice programs are built with a visual drag-and-drop interface. The user selects from a list of available commands and places them in the desired sequence. The programming language provides equivalents for all standard flow of control structures. A simple Alice program is shown in Figure 2.

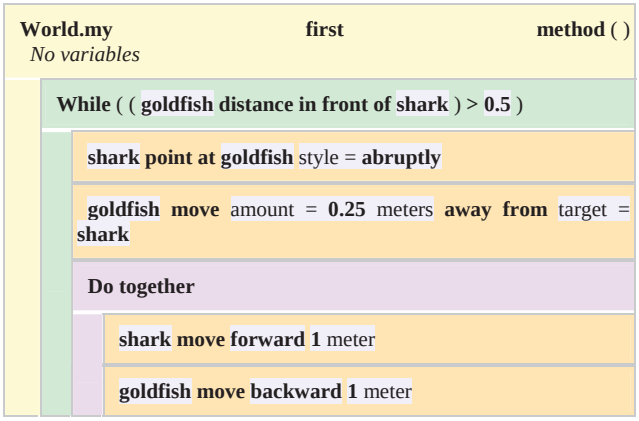

Figure 2. Simple Alice Program

Supporters of Alice cite a number of advantages to its use for teaching introductory programming. First, it is highly engaging. Even new users can easily produce their own

quality 3D animations. Second, like other VPL systems, it greatly reduces the syntactic burden for beginning programmers. The drag-and-drop interface completely eliminates many potential programming errors. Third, Alice maps extremely well to the object-oriented programming model. All characters in Alice are encapsulated objects. Each character has associated information (properties) and things it can do (methods). Character behaviour is controlled by sending messages to the character. All of the traditionally thorny concepts of OO theory occur naturally in Alice, and students reportedly find them easier to grasp in this context than in a text-based programming environment like Java (Goldweber et. al., 2006).

5 Method

In the present study we wished to explore the extent to which students could make the logical connection between a flow of control construct learned in the traditional imperative context and the desired behaviour in an Alice program. The rationale for using systems like Alice to teach programming is that concepts like flow of control "make sense" in the world of Alice. If that is true, we would expect that a student who had 1) learned a particular flow of control construct and 2) wished to produce behaviour in Alice that required that construct, would be able to do so without explicit demonstration.

The study involved two consecutive instances (Semesters 1 and 2 in 2006) of the introductory programming course in a three-year Bachelor of Information Technology degree. The course runs for 12 weeks, and is taught using Borland Delphi Pascal in a console-only environment. Students are usually experienced computer end-users but have no prior formal training in programming. The course is taught entirely procedurally, with no object-oriented theory.[1]

5.1 Course Instance 1 – Semester 1

During the 2006 Semester 1 offering of the course, a new assignment was introduced. Students were to create an animated movie using Alice. Students were given a two introductory sessions with Alice where the basic mechanics of the tool were introduced. They were directed to the provided Alice tutorials but were given no detailed instructions about how to design programs in Alice. They were asked to produce a 2-minute animation in Alice by the end of the term, and were told that their progress would be checked (but not marked) at intervals during the term. No restrictions or requirements were placed on the assignment beyond the 2-minute running time. 85% of students successfully produced the required Alice program.

The rest of the course was a traditional first course in procedural programming. Students were introduced to statements, flow charts, variables, control structures, etc. All course work was done in a text-only console environment. Students were assessed via in-course exercises, exams and assignments. Performance in the course was comparable to previous years, with 76% of the students achieving a passing course mark. Student programming assignments showed acceptable mastery of basic flow of control structures, with 95% displaying correct branching and 80% displaying correct use of loops.

At three intervals during the course (weeks three, six and eight), students were required to submit their Alice programs for checking. The checkpoint intervals were set to follow the presentation in class of a particular flow of control construct by one to three weeks as shown in Table 1.

Construct	Presentation Date	Checkpoint Date
If statements	Week 3	Week 3
While loops	Week 5	Week 6
For loops	Week 6	Week 9

Table 1: Schedule of Alice program checkpoints Semester 1

At each checkpoint, the Alice projects were inspected for the presence of any of the previously presented constructs. The results of the inspection protocol are shown in Table 2. Each cell contains the proportion of student projects that contained the indicated construct.[2]

Students Using Construct in Alice Assignment			
Construct	**3 weeks**	**6 weeks**	**9 weeks**
If statements	0/20 (0%)	0/17 (0%)	1/30 (3%)
Loops	2/20 (10%)	4/17 (24%)	6/30 (20%)

Table 2: Results for Alice program inspections Semester 1

As can be seen from Table 2, the majority of students did not use flow of control structures in their Alice projects. Although student performance on course assessment clearly indicates that they had a good grasp of branching

[1] OO is introduced in the second programming course at our institution. In our experience students benefit from this initial focus on basic syntax and program construction.

[2] Note that several students submitted a final Alice project without having submitted their work-in-progress at the assigned checkpoints.

and looping (95% using correct branching and 80% using correct loop constructs), they did not choose to incorporate these protocols into their Alice programs. Thus, there is only minimal evidence of students spontaneously transferring their understanding of flow of control constructs from the imperative programming environment to the visual environment. We had assumed that, if the Alice environment accurately matches the novice programmer's cognitive model of problem-solving, they would readily transfer all acquired programming knowledge to that environment. This did not occur.

Our assumption may be flawed in two ways:

1. Alice is not actually a more "natural" programming environment. In spite of its direct representation of state and flow of execution, programming in Alice may not actually match the novice's cognitive model of problem-solving more closely than the more traditional programming environment.

2. Alice *is* a more natural programming environment, but the transfer of learning from Pascal to Alice was more difficult than we realised. Our expectation that an understanding of looping would lead to a spontaneous recognition of the opportunities for looping in Alice may have been overly optimistic.

5.2 Course Instance 2 – Semester 2

In the second course instance we attempted to address the notion that the transfer of knowledge from Pascal to Alice may have been "too hard". Perhaps students recognised the value of adding flow of control to their projects, but were unable to do so because of simple mechanical difficulties with Alice. To this end, we repeated the Alice assignment, but increased the amount of explicit instruction given in the use of Alice. Specifically, as each flow of control construct was introduced in the imperative context, its use in Alice was also demonstrated. We anticipated an increase in the use of the constructs in our students' Alice projects.

In addition, the number of checkpoints was reduced, and the time between the introduction of a construct and the associated checkpoint was extended to allow students more time to grapple with this potentially difficult problem. The checkpoint schedule for the Semester 2 course instance is shown in Table 3.

Construct	Presentation Date	Checkpoint Date
If statements	Week 3	Week 4
While loops	Week 5	Week 8
For loops	Week 6	Week 8

Table 3: Schedule of Alice program checkpoints Semester 2

The results of the checkpoint inspections are shown in Table 4.[3]

As expected, a larger proportion of students used flow of control constructs in their Alice project after receiving explicit demonstration of how to do so. It must be noted, however, that the majority of students still do not implement any branching, and a substantial minority do not use loops. Even when issues of mechanical difficulty are mediated, we see only limited transfer from imperative programming to Alice.

Students Using Construct in Alice Assignment		
Construct	**4 weeks**	**8 weeks**
If statements	2/10 (20%)	3/12 (25%)
Loops	4/10 (40%)	7/12 (58%)

Table 4: Results for Alice program inspections Semester 2

6 Discussion

Alice is inarguably a well-constructed, visually compelling, and fun end-user application. However, based on our classroom experience, we question its real pedagogical value for programming education at the tertiary level. Students do not seem to naturally make a strong connection between the formal coding process and what they are doing with Alice. This was demonstrated in their failure to consistently transfer learned programming techniques to their Alice work. It was also indicated by comments collected at the end of the second term, where several students remarked that Alice seemed "simplistic", and to bear little relationship to "real programming".

This view has also been expressed by students at a large tertiary institution in North America (Virginia Polytechnic) who have recently submitted a petition to the school administration requesting the cancelling of Alice teaching. [4] These students maintain that Alice is not an effective tool for teaching programming, even at an introductory level. They claim that Alice is "too simplistic, too difficult to operate and carries little-to-no value beyond a basic conceptualization of object-oriented programming."

These concerns are echoed in recent work by Powers, Ecott and Hirshfield (2007). Powers and her colleagues used Alice extensively in a first programming paper. The first half of the course was taught in Alice, the second half in a high-level object oriented language (once with Java, and once with C++). Powers et. al. recognise the value of Alice in promoting engagement and reducing

[3] The reduced class size is due to this class being a mid-year intake which is consistently smaller than the start of year intake.

[4] http://www.collegiatetimes.com/news/1/ARTICLE/7093/2006-05-02.html

frustration in less-confident students. However, they note that the transition from Alice to a traditional programming language was not smooth. Students were "overwhelmed" by the formal syntactic requirements of the programming language, and "had a difficult time seeing how the Java code and the Alice code related" (pg. 216). In terms of the psychology of programming literature, the mental model of programming that students construct when learning Alice is apparently not well-suited to programming in a traditional context.

Further, Powers et. al. note that even some of Alice's more general benefits of engagement and tractability were lost when students tried to move from Alice to Java or C++. Students became frustrated and lost confidence when they had difficulty with the high-level language, concluding that "their success in Alice had not been 'real programming' but rather just fooling with a toy environment designed for a younger audience" (pg .217).

In the present study we also found that Alice's powerful graphical capabilities may actually be counterproductive when Alice is used a teaching tool, rather than as an animation tool. Feedback indicated that some students may have become distracted by details of the animation process itself and failed to recognise the programming possibilities in Alice. When asked: "What was the most difficult aspect of the assignment?", several students commented on issues associated with narrative and animation not implementation. For example, they identified "creating a perfect walk" and "finding a story" as difficult aspects.

Teaching programming is difficult, and it is natural to seek to ease this difficulty through the development of new teaching tools. Visual Programming Environments like Alice are appealing but we must remember that the eventual goal of programming education is most often to produce programmers skilled in Java or C++, not in Alice. We must be careful that in our quest to make programming easier to teach, we do not wind up not teaching programming at all.

7 References

Adams, J. (2007), Alice, middle schoolers & the imaginary worlds camps, *Proceedings of the 38th SIGCSE technical symposium on Computer science education SIGCSE '07*, 39, 1 307-311

Adelson, B. , Littman, D., Ehrleich, K., Black, J. Soloway, E. (1985) Novice expert differences in Software design, in B. Shackel (Ed*.), Human-Computer Interaction-INTERACT84*, North-Holland.

Brilliant ,S. & Wiseman, T. (1996): The first programming paradigm and language dilemma, *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, Philadelphia, Pennsylvania, United States, 338-342.

Brooks, R. (1978) Using a Behavioral Theory of Program Comprehension in Software Engineering,. *Proceedings of the 3rd International Conference on Software Engineering* New York: IEEE, 196-201.

Brown, M. & Sedgewick, R. (1984), A system for algorithm animation, *Proceedings of the 11th annual conference on Computer graphics and interactive techniques SIGGRAPH '84*, 18, 3, 177-186.

Carlisle, M., Wilson, T., Humphries, J. & Hadfield S., (2005) RAPTOR: a visual programming environment for teaching algorithmic problem solving, *Proceedings of the 36th SIGCSE technical symposium on Computer science education SIGCSE '05*, 37 1 , 176-180.

Cooper, S., Dann, W. & Pausch, R. (2000) Alice: a 3-D tool for introductory programming concepts *Journal of Computing Sciences in Colleges , Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges CCSC '00*, 15, 5, 108-117.

Dann, W., Cooper, S., & Pausch, R. (2000) Making the connection: programming with animated small world , *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education ITiCSE '00*, 32, 3 , 41-44

Davies, S.P. (1990) The nature and development of programming plans, *International Journal of Man-Machine Studies*, 32,4, 461-481.

Davies, S.P. (1993), Models and theories of programming strategy, *International. Journal of Man-machine Studies*, 39, 237-257.

Detienne, F.(1990) Expert Programming Knowledge: A schema-based Approach", in J.-M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore (eds.), *Psychology of Programming*, Academic Press Ltd. 205-222.

Duke, R.. Salzman, E., Burmeister, J., Poon, J. and Murray, L. (2000): Teaching Programming To Beginners – Choosing The Language Is Just The First Step, *Proceedings of the Australasian Conference on Computing Education*, Melbourne Australia, 79-86.

Edwards , A. (1988) Visual programming languages: the next generation *ACM SIGPLAN Notices*, 23 4 , 43-50.

Fix, V., Widenbeck, S, Scholtz, J.. (1993) Mental Representations of Programs by Novices and Experts. *INTERCHI'93 Conference Proceedings*, 74-79.

Giangrande , E. (2007): CS1 Programming Language Options , *Journal of Computing Sciences in Colleges*, 22, 3, 153-160.

Goldman, R. Eguchi, A. & Sklar, E. (2004) Using educational robotics to engage inner-city students with technology *Proceedings of the 6th international conference on Learning sciences ICLS '04*, 214-221.

Goldweber, M., Bergin, J., Lister, R. and McNally, M.F. (2006). A Comparison of Different Approaches to the Introductory Programming Course. In *Proc. Eighth Australasian Computing Education Conference (ACE2006), Hobart, Australia. CRPIT*, 52, 11-13.

Green, T & Petre, M. (1996), Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing*, 7,131-174.

Gugerty, L and Olson, G.M. (1986) Debugging by skilled and novice programmers, *Proceedings CHI'86: Human Factors in Computing systems.*

Hoc, J-M, Green, T, Samurcay & D.J. Gilmore (eds.), (1990) *Psychology of Programming*, Academic Press Ltd.

Hood, C. & Hood, D. (2005) Teaching programming and language concepts using LEGOs, *ACM SIGCSE Bulletin , Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education ITiCSE '05*, Volume 37 Issue 3 19-23

Howell , K. (2003): First computer languages , *Journal of Computing Sciences in Colleges*, 18, 4 , 317-331.

Ichikawa,T. & Hirakawa, M. (1987) Visual programming—toward realization of user-friendly programming environments, *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow ACM '87*, 129-137.

Jeffries, R. (1981), The processes involved in designing software", In J.R. Anderson (Ed.), *Cognitive skills and their Acquisition*, pp. 255-283, 1981.

Kelleher, C. & Pausch , R. (2005) Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers, *ACM Computing Surveys (CSUR)*, 37, 2, 83-137.

Koenemann, J. & Robertson, S. (1991), Expert Problem Solving Strategies for Problem Comprehension. *Proceedings of CHI'91*. 215-130.

Lister, R. (2004) Teaching Java first: experiments with a pigs-early pedagogy, Proceedings *of the sixth conference on Australasian computing education - Volume 30 ACE '04* , 177-183.

Materson, T. & Meyer, M. (2001) SIVIL: a true visual programming language for students *Journal of Computing Sciences in Colleges , Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges CCSC '01*, 16 , 4 , 74-86.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001): A multinational, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**(4):125-140.

McKeown, J. (2004) The use of a multimedia lesson to increase novice programmers' understanding of programming array concepts, *Journal of Computing Sciences in Colleges*, 19, 4, 39-50.

N. Pennington, N. (1987) "Comprehension Strategies in Programming." In Empirical Studies of Programmers: Second Workshop, G. M. Olson, S. Sheppard, and E. Soloway, eds. Norwood, NJ: Ablex. p. 100-113. 1987.

Pattis, R (1994). *Karel the Robot: A Gentle Introduction to the Art of Programming* , John Wiley & Sons, Inc.

Powers, K. Ecott, S. & . Hirshfield , L. (2007) Through the looking glass: teaching CS0 with Alice, *Proceedings of the 38th SIGCSE technical symposium on Computer science education SIGCSE '07*, 39 1, 213-217.

Shneiderman, B. & Mayer, R. (1979) Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Inernational. Journal of. Computing. and Information. Sciences*, 8, 3, 219-238.

Weinberg, G.M. (1971) *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Wester, F., Sint, M., & Kluit ,P. (1997) Visual programming with Java; an alternative approach to introductory programming, *ACM SIGCSE Bulletin , Proceedings of the 2nd conference on Integrating technology into computer science education ITiCSE '97*, 29 , 3 , 57-58.