Learning Resources and Tools to Aid Novices Learn Programming

Stuart Garner
Edith Cowan University, Perth, Australia

s.garner@ecu.edu.au

Abstract

It is well known that learning introductory software development is a difficult task for many students. This paper discusses some of the resources and tools that are available, or have been experimented with, that might be of interest to instructional designers of programming.

The resources and tools are discussed in the context of the four phases of the software lifecycle, these being: analyse the problem; design and develop a solution / algorithm; implement the algorithm; and test and revise the algorithm. The tools that are discussed include microworlds, videoclips, flowchart interpreters, and program animators.

Keywords: novice programming; software lifecycle; programming tools.

Introduction

It is well known that learning to program is a difficult and frustrating process. Novice programmers must learn concepts and skills that often bear little relation to their past experiences (Smith & Webb, 1999). This aim of this paper is to discuss some of the resources and tools that are available, or have been experimented with, that may prove of use to the instructional designers of programming courses of study. The paper begins by discussing a learning framework for instructional designers and how that might be applied to the learning of programming. The four phases of the software lifecycle in which students have to construct knowledge are introduced and the body of the paper discusses the resources and tools that are available, or have been experimented with, in the context of those lifecycle phases.

Learning Framework

A useful starting point for the discussion concerning resources and tools is to consider a learning framework that has been put forward by Oliver (1999) and that is shown in Figure 1. The framework comprises learning activities, resources and supports. In the context of learning programming, the learning activities are the tasks which students are expected to participate in to help them learn. These might include solving a problem, designing some pseudo code, implementing an algorithm in a programming language, or testing a program. The learning activities play a fundamental role in determining learning outcomes (Wild, 1997) and they determine how learners engage with the various materials.

Learning resources provide the content for a course and can be thought of as the materials that are used to help students construct their knowledge and meaning with respect to a domain of knowledge. Traditionally these resources have been available in the form of books and lecture

Paper Accepted as a Regular Paper

notes and the move to flexible technology based systems has led to a lot of content being made available electronically. Most programming courses are still usually underpinned by a textbook although increasingly there are online tutorials, quizzes, simulations etc.

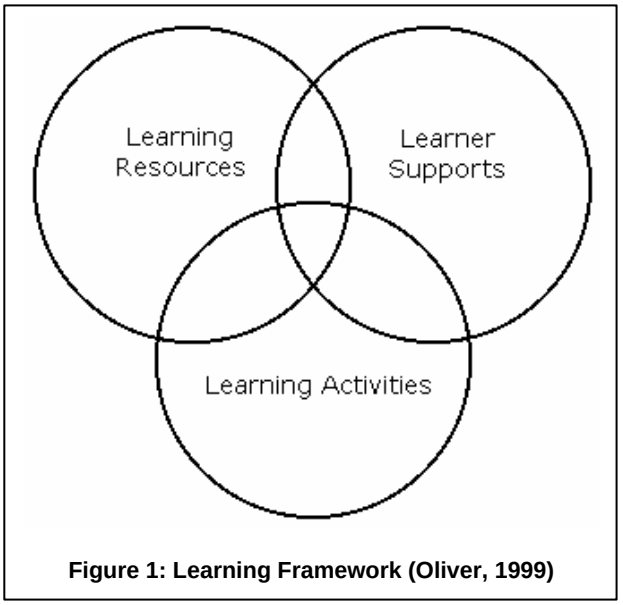

Figure 1: Learning Framework (Oliver, 1999)

Learning supports are the third element of the framework and can be thought of as the supports required to help guide and provide feedback to learners in a way that is responsive and sensitive to learner individual needs (McLoughlin, 1998). In "traditional" settings such supports have been provided by actively involved teachers (Laurillard, 1993) whereas in technology based learning environments, such supports are often known as "scaffolds" to help learners during their knowledge construction process (Roehler, 1996). In programming, an example of such a support is the facility that some programming editors have to help complete lines of programming code for the user as they are keyed-in. Frequently, such supports are provided in the form of software tools.

Software Lifecycle

Learning to program involves constructing knowledge in the four phases of the software lifecycle, these being:

- Phase1: Analyze the problem
- Phase2: Design and develop a solution / algorithm
- Phase3: Implement the algorithm
- Phase4: Test and revise the algorithm

All of the above activities need to be learned by novices and each can be considered as a learning activity or task within Oliver's learning framework. This paper will now consider each of the individual tasks and discuss some of the resources and tools that might be used to help novices with these tasks.

Phase 1: Analyse the Problem

The standard way of teaching and learning programming has not changed very much over the years. Usually a teacher introduces a new control structure or data structure, shows some examples to students, and then expects students to be able to solve problems that are either novel or possibly similar to those that he or she has demonstrated. It appears that very often syntax is being taught at the expense of problem solving. Students who experience this form of instruction frequently complain that although they might understand and follow the teacher's reasoning during a demonstration, they find it very difficult to then solve a problem on their own.

Available resources to help instructional designers are limited in this area, however three that have been identified are the tool SolveIt; the use of video clips; and the use of microworlds.

SOLVEIT

SOLVEIT is a prototype of an integrated environment to support students learning programming. Support is provided through all problem solving stages, including formulating the problem, planning and designing a solution, and testing and delivering that solution (Deek & McHugh, 2000). The interface is shown in Figure 2. In this section, the area of interest is problem analysis and three tools are available in SOLVEIT to help with problem formulation, these being: the problem description editor; the verbalisation tool; and the information elicitation tool.

Figure 2: SOLVEIT (Deek & McHugh, 2000)

The problem description editor allows a student to enter and save the problem statement into the system within a multiple-view reference database. The verbalisation tool allows questions to be presented to students while the problem statement is visible in the editor. A student's answers to the questions are saved in a project notebook together with subsequent verbalisation sessions. The transcript of such recordings is then one of the project's deliverables.

The aim of this tool is to help students think more deeply about the problem in question. In addition to the project notebook, SOLVIT also provides a graphics editor that permits students to make and save drawings / sketches concerning the problem. The information elicitation tool is used to extract relevant information from within the problem description. This information includes goal, givens, unknowns, conditions and constraints and is stored within the multiple-view reference database.

SOLVIT is one of the few specific tools available to support the teaching and learning of programming that provides help in this important area of problem analysis and formulation. However, to date it does not appear to have been evaluated.

Video Clips

Multimedia Command Centre

A multimedia command centre (Garner, 1997) was produced and was then utilised to build a Visual BASIC programming tutor. The tutor allows the delivery of video clips in the form of e-movies that are recordings of Windows sessions together with audio narratives. The e-movies can be replayed on a PC and the interface is shown in Figure 3. The course includes sets of programming problems and students are helped with problem analyses by being able to watch e-movies that give extensive hints and tips on how the problems might be tackled. Students indicated during interviews that these movies had proved extremely valuable and that they believed that they had helped their problem solving abilities.

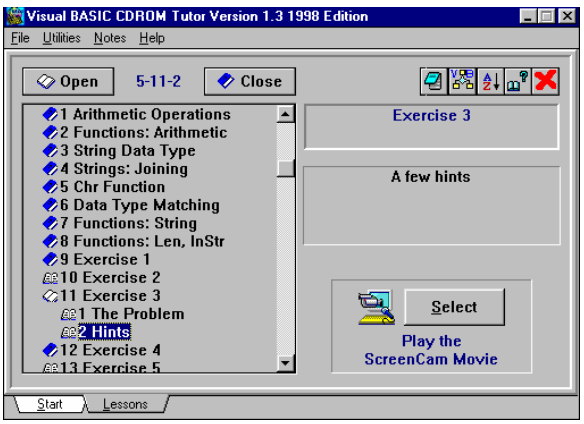

Figure 3: VB Tutor

Production of Video Clips with a Pen Mouse

Crown (2002) describes his method of creating e-movies that walk through the process of analysing engineering problems. A real-time screen capturing program together with pen mouse and tablet are used enabling him to "hand-write" his problem analyses whilst making explicit his thought processes via an audio recording. Although the problem domain is engineering, the method is directly transferable to programming problem analyses. Students were overwhelmingly positive in there comments as the movies could be replayed on demand.

Microworlds

A method of reducing the cognitive load on students who are learning programming is to use microworlds. Such systems provide a very narrow range of problems in which control structures are emphasised over data structures. This reduces the burden on students in the problem analysis stage. Examples of such systems include Karel the Robot (Pattis, 1995); RoboPascal (Carey, 1996); and Squeak (Kaye, 2002).

The most popular microworld program for learning programming is Karel and this has been around since the 1980s in various formats. Problems can be set for students by defining a microworld as shown in Figure 4 and then setting a task to be achieved by the robot. The world includes streets and avenues; walls; beepers that can be picked up or put down; and the "robot" which is depicted by an arrow.

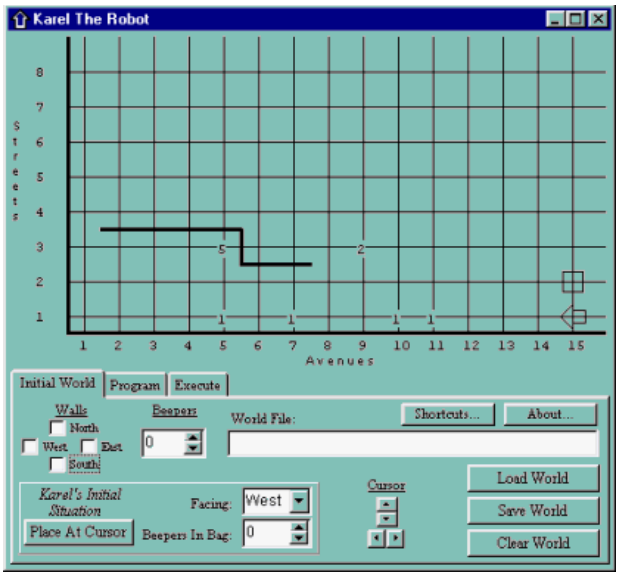

Figure 4: A Simple Karel World

The web site of a high school in Australia (Alex Hills, 2002) has some well thought-out tutorials and assignments for students concerning Karel. Particularly useful are the hints and tips that are given to help students analyze the problems that have been set.

Phase 2: Design and Develop a Solution / Algorithm

After problem analysis it is necessary to design a solution. This involves students creating an algorithm in some format that will hopefully solve the problem that they are attempting. Non-technological tools that were often used in the past included hand-drawn flowcharts, hand-drawn Nassi-Schneiderman diagrams, and pseudo code. Often, programming teachers omit using such tools, and algorithms are entered directly in a programming language by students. This of course makes the learning of programming even more difficult for students as program design and the syntax of a language become intertwined.

The following are some of the resources and tools that are available to help students design and develop solutions / algorithms to problems. SOLVEIT is not included, as it appears that the latter parts of the software lifecycle have not yet been implemented in the prototype.

Karel the Robot

Karel has a very simple underlying language that only has five primitives, these being: move; turnleft; pickbeeper; putbeeper; turnoff. Other instructions can be defined using the primitives. Because the language is so simple and only control structures are emphasised, algorithms are keyed-in directly to the

system by students as shown in Figure 5. For Karel, the language itself can be considered as a form of pseudocode.

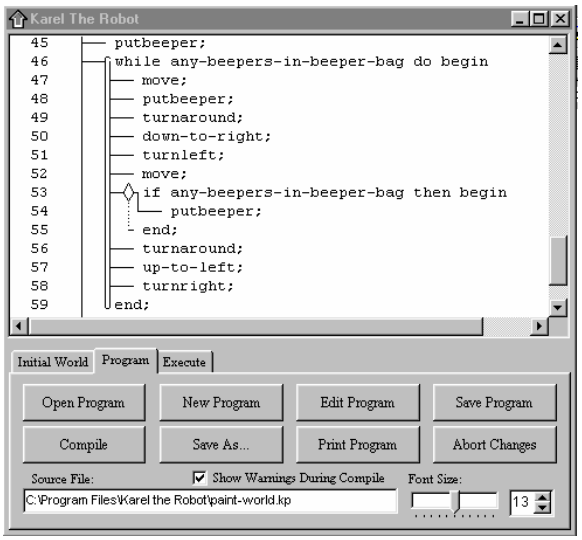

Figure 5: An algorithm in Karel

Flowchart Interpreter Program: FLINT

FLINT (Crews & Ziegler, n.d.) provides a visual algorithmic design environment that utilises flowcharts. The system removes the focus from the syntactic details of a programming language by providing students with an iconic interface for developing flowcharts as shown in Figure 6. The point-and-click interface hides low-level details from the user and frees the students to concentrate on designing the algorithm to solve a given problem.

However, Crews and Ziegler make the point that removing the focus on syntax does not mean that students will focus on more appropriate issues. They have therefore provided a structure chart diagrammer that students must use before being allowed to start working on a flowchart. Each of the steps developed in the structure chart can be implemented by a separate flowchart.

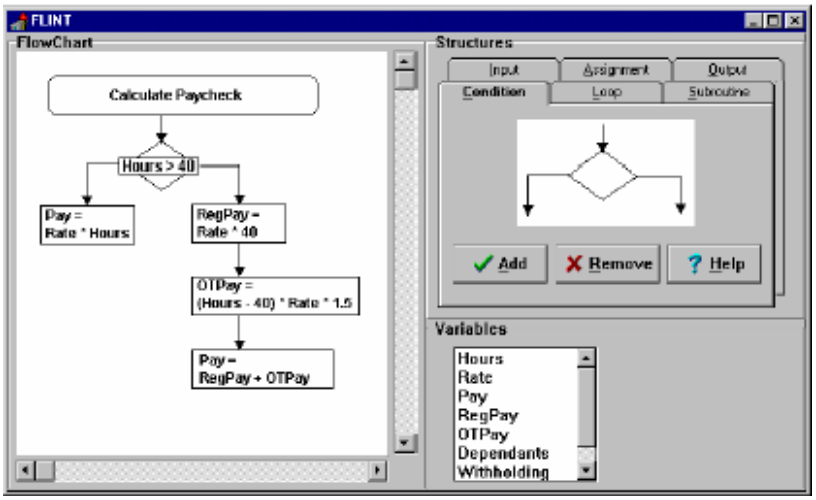

Figure 6: A Flowchart in FLINT

Hand-drawn flowcharts fell out of use mainly because they were so difficult to update. It was always recognised that they were very useful as a way of providing visualisation of an algorithm. Graphics tools such as FLINT now provide the facility to maintain such flowcharts.

Tools to Animate Fundamental Algorithms

When designing an algorithm for a problem, students may have to make use of or amend what might be termed a "fundamental algorithm". Such algorithms include: sorting data in an array; searching for a data item in an array or a file; or merging two sequential files. Programming teachers usually use traditional "talk and chalk" techniques to explain these algorithms, however there are visual tools available to help improve student understanding. For example, Hansen et al (1998) suggest that an experiment that they undertook provides preliminary statistical validity to the conclusion that hypermedia visualizations, or animations provided in context, are more effective than textbooks.

There are many such examples in the form of Java applets and which are available on the Internet. One example is "The Sort Algorithm Animator V1.0" (Ploedereder, 2000) and the interface is shown in Figure 7. When the animation is running, the heights of various "bars" are compared and those bars are

swapped if necessary. The animation can be stepped through or run automatically, with the speed being varied.

An algorithm animation actually serves two fundamental purposes. It provides a concrete depiction of the abstractions and operations inherent in an algorithm or program, and it portrays the dynamics of a time-evolving process. (Byrne et al, 1996).

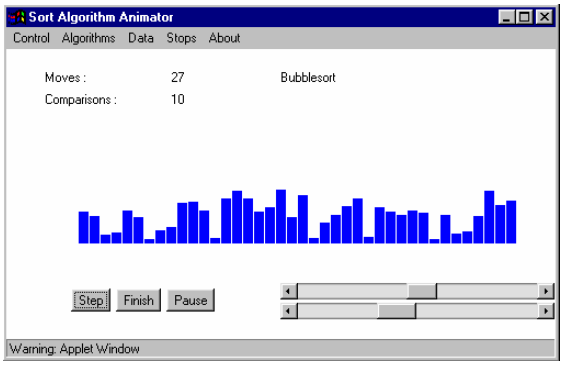

Figure 7: Sort Animator

Phase 3: Implement the Algorithm

The third element of the software lifecycle is "implement the algorithm" and requires students to convert their algorithms to executable programming code. Microworld systems, such as Karel the Robot, have algorithms written directly at the program design stage, as the design language is the same as the implementation language. The flowchart tool FLINT provides support for phase 3 and so too do many programming development environments.

FLINT

It was seen earlier that FLINT allows students to develop flowcharts in order to design algorithms. As can be seen in Figure 6, it also allows variables to be defined and assignment statements to be stipulated, and when this has been done, a flowchart can be "executed" step-by-step. This becomes very important in the "Test and Revise" stage of the software lifecycle as will be seen later.

Programming Language and Environment

Normally a program design or algorithm is implemented directly in conventional programming code and the programming language and environment that are utilised have a big effect on student learning. The integrated development environment (IDE) can act as a support and tool for students and yet most have been designed for professional programmers and have a very steep learning curve thereby increasing the already high cognitive load on students. Also, many programming teachers have different ideas on what is the "right" language to teach as a first programming language which leads to "language wars" in many computer science and information systems departments. In a recent survey in Australia the top two languages that were used in universities were Java (23 institutions) followed by Visual BASIC (14 institutions) (De Raadt, 2002).

An ideal situation would be to use a specialised teaching language that illustrates most of the required control and data structures however, although we as educators recognise the pedagogical value of this, the majority of students do not and prefer to use a language that is "out there" in the market place. Unfortunately, such conventional languages are becoming more complex. For example Visual BASIC version 6 is now to be superseded by Visual BASIC .NET and the complexity of the language has become a real challenge for students. Perhaps it is the time to take a step back, persuade students of the benefits of a "teaching language" and introduce something like Liberty BASIC (Gundel, n.d.) which is more straightforward and is similar to the original Visual BASIC program that was introduced in 1991.

The kinds of supports provided by an IDE to help a student implement an algorithm include:

- Immediate syntax checking of statements.
- Help in completing statements via intelligent editors. For example, if the following were entered in the Visual BASIC editor:

Let initial = Left(

then the syntax of the Left function would be displayed.

- The automatic filling-in of procedure headers and footers.

However, as IDEs have been created to help professional programmers, the help systems are often incomprehensible to most students.

One particular teaching language environment that is gaining popularity in universities is BlueJ (BlueJ, 2002). This is used specifically for the learning of object oriented programming and was developed at Monash University, Australia. An example of a quote from a user that is on the BlueJ web site is:

> *I find that BlueJ removes all the complications of other tools and replaces them with features that let me concentrate on my education rather than the tool itself.*

The quote reinforces the fact that as educators it is very important that tools are selected or built that help rather than hinder students.

Phase 4: Test and Revise the Algorithm

Students learning and understanding of how an algorithm executes is supported during the testing phase of the software lifecycle. This area can be crucial in improving student mental models of how control and data structures function. Resources, supports and tools range from the built-in debuggers that many IDEs have to special tools that can animate a student's program.

Conventional Debugger

Conventional IDEs, such as Visual BASIC, have built-in debuggers that allow programs to be stepped through line-by-line and the contents of variables to be displayed. Students find this mechanism particularly useful in improving their understanding of how their algorithms work and obviously in fixing up problems. In the author's view, students need to be encouraged, by careful instructional design, to use this important tool from the very beginning of a course in order to gain an understanding how example algorithms work that are presented by their tutor.

Program Animators

In the section concerning phase 3, the design of algorithms, tools to help animate fundamental algorithms were discussed. These are demonstration tools that only work on one class of algorithm such as sorting. Other tools have been developed to animate any algorithm in a particular programming language and these are called program animators. They help students visualise program execution and they promote low-level models of programming. Animators reinforce a model of program execution by explicitly showing how the execution of a statement affects the program state and environment in which the following statement is executed (Smith & Webb, 1998).

Two examples of such visualisation tools are BRADMAN (Smith & Webb, 2000) and VINCE (Rowe & Thorburn, 2000). BRADMAN is a glass-box interpreter that helps students in their learning of the C programming language. In addition to the features of "standard" debuggers, it also contains a variables display; a verbal explanation of each statement as it is executed; and more visible input / output facilities. An example interface is shown in Figure 8. In an evaluation, it was found that a student group that had used BRADMAN performed significantly better than a control group at the manual interpretation of programs.

VINCE is also used as a tool to help in the teaching and learning of C programming. It has been written entirely in Java and is therefore accessible as an applet on a Web page. It appears to possess similar features to BRADMAN including a memory map so that variable contents can easily be inspected. In its evaluation, the use of VINCE did not change the students' perceptions of their programming ability relative to those in a control group however their performance on a series of programming questions was better.

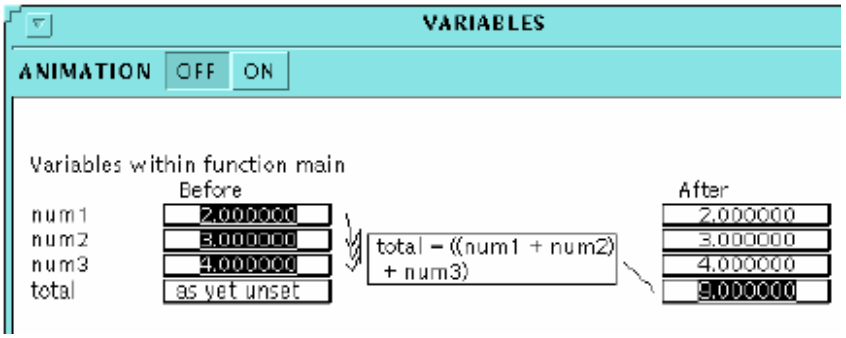

Figure 8: Bradman Interface (Smith & Webb, 1995)

FLINT also provides an animation mechanism for students. The sequential nature of programs becomes literally visible as every flowchart can be executed step-by-step. The statement that is currently interpreted is highlighted and control passes to the next statement at the student's command. In addition, FLINT makes variable values observable, allowing the student to follow the effects of the program state-statements (Crews & Ziegler, n.d.).

Karel the Robot and Animation

Because Karel is a purpose-built tool to help students learn programming, the trace and animation facilities are built-in and very strong. Not only can students view the animation of the code as shown in Figure 9, but they can also view the movement of the robot, depicted as an arrow in Figure 10, as each line of code executes. This dual view helps students in their construction of knowledge concerning control structures and simple algorithms.

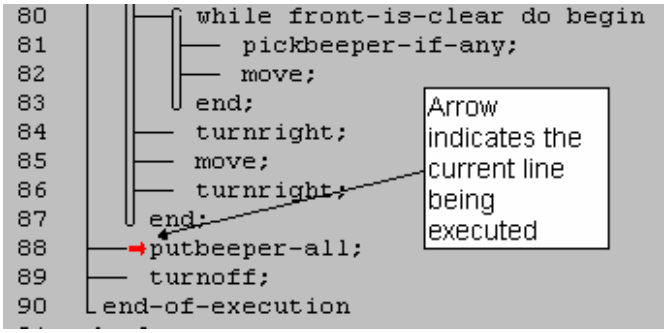

Figure 9: Animation of Karel Code

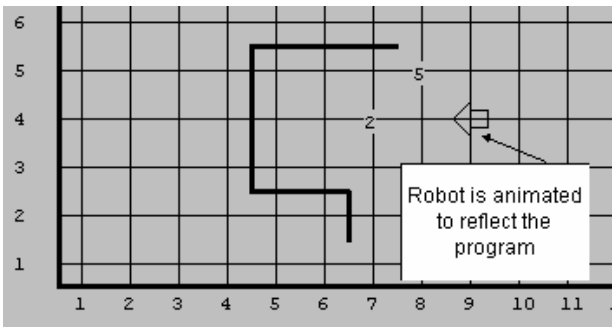

Figure 10: Animation of Karel the Robot

Discussion

This paper has discussed just some of the resources and tools that are available, or have been experimented with, to help students in their difficult task of learning to program. They were discussed in the context of the four phases of the software lifecycle and certain phases are more strongly supported than others.

Phase 1, analyse the problem, lacks good support for the analysis of "conventional" programming problems. The author suggests that instructional designers and teachers concentrate their efforts on this phase as it is an area that students find particularly difficult. Perhaps the simplest approach is to build up video clip resources that clearly show how experts analyse problems.

Phase 2, design and develop a solution / algorithm, is often merged with phase 3, implement the algorithm, with students entering programming code directly. However, FLINT seems to offer a very useful approach as control structures can be entered directly in a graphical format in the form of flowchart symbols.

Phase 3, implement the algorithm, is usually carried out in a conventional programming language, however FLINT again could offer a useful alternative approach as variables can be defined and assignment statements stipulated through the FLINT interface. A shortcoming that might be levelled at FLINT is that it is not powerful enough to support the full range of programming constructs such as array processing and parameter passing.

Phase 4, test and revise the algorithm, is well supported with conventional debuggers and program animators. However the major problem is that the animators that have been built for specific programming languages, for example BRADMAN and Vince, only supporting C programming. It would be useful to have educational tools that supported the animation of the popular programming languages such as Visual BASIC and Java.

It was also seen that the microworld of Karel the Robot looks particularly attractive as it gives strong support through phases 2 to 4 and phase 1, problem analysis, is less of an issue as the problems are in a narrow domain. However, the time constraints of a 13-week introductory course in programming may preclude the use of such a microworld prior to the introduction of a "conventional" programming language.

For students to be successful in learning to program, all four phases of the software lifecycle need to be well supported with appropriate resources and tools. Clearly this is still not the case and more work and research will need to be carried out in this area.

References

Alex Hills SHS (2002). *IPT @ Alex Hills*. Retrieved from the World Wide Web 16 Oct 2002 http://www.alexhillshs.qld.edu.au/faculties/resources/ipt/index.htm

BlueJ Website. Retrieved from the World Wide Web 2 Oct 2002 http://www.bluej.org

Byrne, M. D., Catrambone, R., & Stasko, J. T. (1996). *Do Algorithm Animations Aid Learning?* Retrieved from the World Wide Web 17 Oct 2002 http://citeseer.nj.nec.com/byrne96do.html

Carey, D. (1996). *Teaching Algorithms and Programming Concepts Using an Object-Oriented Language*. Paper presented at the Australian Conference in Computer Education 96, Brisbane.

Crews, T., & Ziegler, U. (n.d.). *The Flowchart Interpreter for Introductory Programming Courses*. Retrieved from the World Wide Web 14 Oct 2002 http://fie.engrng.pitt.edu/fie98/papers/1107.pdf

Crown, S. W. (2002). *The Development and Use of Tutorial Movies using a Pen Mouse in an Engineering Problems Based Course*. Paper presented at the ED-MEDIA 2002, Denver, Colorado.

De Raadt, M., Watson, R., & Toleman, M. (2002). *Language Trends in Introductory Programming Courses.* Paper presented at the Informing Science 2002, University College Cork, Ireland.

Deek, F. P., & McHugh, J. A. (2000). *Prototype Tools for Programming.* Paper presented at the EdMedia 2000, Montreal, Canada.

Dehoney, J., & Reeves, T. (1999). Instructional and social dimensions of class web pages. *Journal of Computing in Higher Education, 10*(2), 19-41.

Garner, S. (1997, December 1997). *Cost Effective Interactive Multimedia with Lotus ScreenCam and a Multimedia Command Centre.* Paper presented at the International Conference in Computers in Education 97, Kuching, Malaysia.

Gundel, C. *Liberty BASIC*. Shoptalk Systems. Retrieved from the World Wide Web 21 Oct 2002 http://www.libertybasic.com

Hansen, S., Schrimper, D., & Narayanan, N. (1998). *From Algorithm Animations to Animation-embedded Hypermedia Visualizations*. Retrieved from the World Wide Web 21 Oct 2002 www.eng.auburn.edu/departments/cse/research/vi3rg/vi3rg.html

Laurillard, D. (1993). *Rethinking University Teaching: A Framework for the Effective use of Educational Technology*.: London Routledge.

McLoughlin, C., & Oliver, R. (1998). *Scaffolding Higher Order Thinking In A Telelearning Environment.* Paper presented at the Ed-Media/Ed-Telecom 98 World Conference On Educational Multimedia And Hypermedia & World Conference On Educational Telecommunications, Virginia.

Oliver, R. (1999). Exploring strategies for on-line teaching and learning. *Distance Education*, *20*(2), 240-254.

Pattis, R. E. (1995). *A Gentle Introduction to the Art of Programming* (2nd ed.): New York, Wiley.

Ploedereder, E. (2000). *The Sort Algorithm Animator V1.0*. Retrieved from the World Wide Web 21 Oct 2002 http://www.informatik.uni-stuttgart.de/ifi/ps/Ploedereder/sorter/sortanimation2.html

Roehler, L. R., & Cantlon, D. J. (1996, May 10th 1996). *Scaffolding: A Powerful Tool in Social Constructivist Classrooms*. Retrieved from the World Wide Web 3 May 1998 http://www.educ.msu.edu/units/literacy/paperlr2.htm

Rowe, G., & Thorburn, G. (2000). VINCE - an on-line tool for teaching introductory programming. *British Journal of Education Technology*, *31*(4), 359-370.

Smith, P. A., & Webb, G. I. (1995). *Reinforcing a Generic Computer Model for Novice Programmers.* Paper presented at Ascilite 1995, Melbourne.

Smith, P. A., & Webb, G. I. (1998). *Overview of a Low-level Program Visualisation Tool for Novice C Programmers.* Paper presented at the International Conference on Computers in Education '98, Beijing, China.

Smith, P. A., & Webb, G. I. (1999). *Evaluation of Low-Level Program Visualisation for Teaching Novice C Programmers.* Paper presented at the International Conference on Computers in Education '99, Tokyo, Japan.

Smith, P. A., & Webb, G. I. (2000). The Efficacy of a Low-Level Program Visualisation Tool for Teaching Programming Concepts to Novice C Programmers. *Journal of Educational Computing Research*, *2*(2-2000), 187-215.

Wild, M., & Quinn, C. (1997). Implications of educational theory for the design of instructional multimedia. *British Journal of Educational Technology*, *29*(1), 73-82.