Teaching Programming

The State of the Art

Eileen Costelloe,
Department of Computing,
Institute of Technology Tallaght,
Dublin 24.
E-mail:eileen.costelloe@it-tallaght.ie

“The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relation or truths. Its province is to assist us in making available what we already are acquainted with.”

Ada Augusta, Countess of Lovelace (1815-1852)
Table of Contents

1. Abstract ............................................................................................................... 3

2. Introduction ........................................................................................................ 4

3. Classification of approaches .............................................................................. 7
   3.1 Lectures and labs ............................................................................................. 8
   3.2 Software Visualization .................................................................................. 12
      3.2.1 Program visualization ............................................................................. 15
      3.2.2 Algorithm animation .............................................................................. 32
      3.2.3 Visual programming .............................................................................. 44
   3.3 Robots ............................................................................................................ 50
   3.4 Problem based learning .................................................................................. 58
   3.5 Cognitive apprenticeship .............................................................................. 66
   3.6 Miscellaneous ............................................................................................... 70

4. Educational theory .............................................................................................. 73

5. Discussion ........................................................................................................... 76

6. Conclusions ......................................................................................................... 87

7. References .......................................................................................................... 89
Abstract
Teaching programming to novices at third level has proven to be a challenge for both staff and students. Many students find the programming module difficult and disheartening and this could have an impact on their attitude to software development throughout the course and as a career choice. For staff involved in teaching programming it can also be very disheartening when students apparently fail to understand and be able to apply even the basic constructs. These difficulties have prompted researchers to investigate tools and approaches that may ease the difficulty of teaching and learning programming. These approaches are categorised and discussed in this report in terms of the motivation that prompted their development, examples of their use and proposed benefits. An overview of educational theory and learning styles is included. The report discusses the relative merits of the different approaches and concludes that while empirical evidence has indicated the merit of certain types of algorithm animations, without empirical evidence on other approaches it is difficult to infer the effectiveness of these approaches.
Introduction
Frequently attributed with being the first computer programmer, the Countess of Lovelaces comments still apply to the process of solving problems on a computer. A program is a set of instructions for a computer to follow and the programmer formulates this set of instructions in order to solve some predefined problem. Data is what we conceptualise as the input to the program, the computer then follows some process to generate a predefined output.

In order to write a program two phases are involved, the problem solving phase and the implementation phase. Linn & Clancy (1992) found that “for programmers to develop competency, they need to have good problem solving skills and a thoroughly organized knowledge of a programming language”. In the first phase a solution/design is generated to solve the problem and in the implementation phase the proposed solution is translated into a programming language. Here we see the marriage of two different skill sets that are required to write a program, problem solving and implementation.

Problem Solving
Riley (1981) concluded that many students entering college have problem-solving skills that are “woefully inadequate”. Henderson (1986) notes that problem solving and analytical thinking are students’ major weaknesses in a computer science course and that a major theme of a computer science course would be to emphasise these skills. Henderson (87) argues that one of the keys to effective design-oriented problem solving is learning to use abstraction, by providing students with tools for thinking abstractly, such tools, e.g. representative like data flow, and basic mathematical concepts like sets and functions are some of the foundations which must be established early in the curriculum. He argues that the ideal computer science curriculum student should learn fundamental computer science concepts based on mathematical foundations and be able to use these concepts in the analysis and development of software.

Masheshwari (1997) states that programming is a study in clear thinking and problem solving and that it gives students practice at building representations and working in a methodical manner. He argues that programming fosters problem solving through a top-down approach, whereby large problems are separated into manageable components that are solved individually and then
assembled into the correct solution to the problem. Programming encourages learners to evaluate their solutions and thinking process and therefore this cognitive process allows the student to transfer newly acquired problem solving skills to novel problem situations.

Whatever approach to problem solving is adopted, it is recognised that this is a necessary and first step in the development of software.

**Implementation**

The second phase that is involved in learning to program is the implementation phase, in other words writing the proposed solution in a specific programming language and executing and debugging the code in a specific environment. This phase involves a number of areas that the student must master:

- **Syntax of the chosen language**, Students often spend large amounts of time trying to debug syntax errors and coming to terms with a new syntax. The absorption of all the syntax rules for a small program sometimes seem to overwhelm them to the extent that they focus more on the syntax rather than the design of their solution. They are often overjoyed at the fact that the program compiled error free regardless of the fact that their answer is incorrect. Linn & Clancy (1992) argue that instruction that focuses on the syntax of a programming language will not impart the skills required to become an effective programmer, and they argue that introductory textbooks encourage this kind of instruction due to emphasising many different programming techniques under page constraints.

- **Programming constructs**
  Even the most basic of programming constructs, such as sequence, iteration and selection can cause a lot of problems for novice programmers and where they state that they can understand an example worked through in a classroom environment, when they come to apply the principle to a new problem area they cannot make the connection. When they progress to simple data structures such as arrays and linked lists they often feel that they are overwhelmed and can start to feel that maybe they have chosen the wrong course.

- **Development environment**
  In order for the students to edit, compile, execute and debug their programs they must become familiar with a development environment, which in itself can often lead to added problems of
confusion. The integrated development environments (IDEs) has several disadvantages for
novice programmers, for example, problem-solving and design are not supported, and the
debugging features are aimed at experienced programmers, leaving novice programmers
overwhelmed (Ziegler & Crews 1999).

- Testing and debugging
  The lack of feedback from some development environments leave the student confused when
their program does not work as expected and they are faced with cryptic error messages to
decipher. The generation of effective test data and test plans add another layer of complexity
for novice programmers.

Given the areas with which the novice programmer must become familiar it is of no great surprise
to know that students find learning programming difficult. “The graduating student who professes
a complete inability to write a simple program is commonplace “(Jenkins 2001).
Classification of Approaches

A number of approaches and tools have been developed or are being used for teaching programming and these can be categorised as follows:

- Lectures and Labs
- Software Visualisation
  - Program Visualisation
  - Algorithm Animation
  - Visual Programming
- Robots
- Problem Based Learning
- Cognitive Apprenticeship
- Miscellaneous
Lectures & Labs

According to Boyle et al (1998) “traditional approaches to teaching programming” involved “a blend of lectures, reading and practical sessions”. The lectures traditionally will cover a concept which will be supported by reading material, either lecturer-generated or a text reference, and practical exercises in a laboratory session which aims to put into practice what has been covered in the pre-ceding lectures. The lecturer’s role has traditionally been descriptive, covering concepts and syntax of a particular language to large groups, concentrating on subject content. The lab sessions are problem-based with the students being presented with problems to which they must design and implement solutions in a given programming language. Typically this approach is content driven with predefined problems on which the students work. Within the lecture environment the students are, in general, passive recipients of the information, with minimal interaction, particularly when large groups are involved. They then tend to work individually in the lab sessions to solve the weekly problems and assessment is traditionally on an individual basis. The classroom environment, large numbers, in which they receive lectures is not contusive to interaction and does not take into consideration the varied backgrounds and abilities of the students, the current classroom model is, according to Holmes et al (2001) ”a product of the industrial revolution where groups of students of the same age come to a single physical location to be instructed in the same subject matter at the same pace”. The lecture method has been criticised for centuries, Samuel Johnson argued that lectures were “unnecessary” as all could read and books were numerous. It has been called an anachronism, imposing passive roles on students and being unadaptive to individual student’s needs (Gage & Berliner, 1992). However those in favour of the lecture method will point to its flexibility and the reinforcement it offers students. While research has shown that it can be an effective teaching method under certain conditions Gage and Berliner (op. cit.) outline the circumstances where lecturing is not appropriate:

- Objectives other than the acquisition of information is sought
- Long term retention is necessary
- The material is complex, detailed or abstract
- Learner participation is essential to achievement of objectives
- Higher-level cognitive objectives (analysis, synthesis, evaluation) are the purpose of instruction
Given these research findings and the fact that they partially mirror the objectives of teaching programming one could conclude that the traditional lecture mode of teaching does not suit teaching programming.

It is an accepted fact the students find programming difficult (Smith and Webb, 2001) and much work is underway to move from this traditional approach to teaching programming to a more active and engaging mode of teaching. According to Boyle et al (1998) “traditional approaches to teaching programming”...“typically lend to a number of problems:

- Passive learning styles
- Over emphasis on language form as opposed to function
- Premature complexity
- Premature abstraction

Traditional teaching approaches imposes a rule based approach on the learner and restrict their natural problem solving skills”

According to Linder (Linder et al op. cit.), “higher education has an over reliance on traditional didactic methods of teaching” and they presented Magnesen’s table which gives average percentages of material retained in long-term memory based on the modality of interaction:

<table>
<thead>
<tr>
<th>Modality of Interaction</th>
<th>Percentage retained long term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading</td>
<td>10%</td>
</tr>
<tr>
<td>Hearing</td>
<td>20%</td>
</tr>
<tr>
<td>Seeing</td>
<td>30%</td>
</tr>
<tr>
<td>Seeing and Hearing</td>
<td>50%</td>
</tr>
<tr>
<td>Discussing</td>
<td>70%</td>
</tr>
<tr>
<td>Doing and Discussing</td>
<td>90%</td>
</tr>
<tr>
<td>Teaching and tutoring</td>
<td>95%</td>
</tr>
</tbody>
</table>

_Table 1: Magnesen’s percentage material retained based on modality of interaction_
The importance of “Doing and Discussing” and “Teaching and Tutoring” has forced a re-evaluation of how we teach programming with new and innovative approaches being developed, which are a variation on the traditional approach.

Such a re-evaluation was conducted at the University of Queensland in 1998 when it was decided to move to Java as the language for one-semester first year programming subject (Duke et al, 2000). The team decided that the aim of the course should be an exploration of the art and pleasure of programming by taking an experiential approach, wanting the student to explore, experiment, experience and extend. To this end they developed a course based on problem-based learning, flexible learning and the inverse curriculum. The concepts were introduced first in problems, encouraging students to question and explore, and then discussed and developed, a reversal of the traditional approach. The problems, varying from easy to moderately difficult, were made available on the subject’s web page for independent and self-paced study, and came with hints leading to solutions. In this way the students are empowered to take control of their studies, and the role of the staff then becomes supportive rather than descriptive. By using an inverse curriculum approach, where the students make use of existing classes from the beginning to create relatively sophisticated applications, the students understanding is deepened and their interest stimulated. This approach had an impact on the use of the lectures and labs where the lecture sessions involved the active creation and running of code, exploratory and discursive. The lab sessions became tutored with individual assistance.

While the Lecture & Lab format still remains the prevalent approach to teaching programming, more innovative staff are looking at integrating new tools and techniques into the curriculum, in either the lectures or the labs in order to enhance the learning experience of the students.

**Benefits of Lectures & Labs**

- Provides a guided learning experience for novice students
- Labs provide experience for the students to practice their newly acquired knowledge
- Provides reinforcement
- Lectures provide an introduction to an area which is developed through lab exercises
- Lectures are necessary to arouse interest in the subject
- It is cost-effective and flexible
Lecture and Lab - Pedagogy

The traditional lecture approach adopts a behavioural theory to learning; the main behavioural theory being applied is operant conditioning where the student learns as a result of reinforcement, be it positive or negative. Students may be reinforced by the lecturer’s humour, drama, enthusiasm and the knowledge and comprehension they acquire. The lecture situation may also provide social reinforcement and emotional reassurance (Gage & Berliner, op. cit.). The combination of lecture and labs moves towards a constructivist model where the students are using the lab exercises to construct their own knowledge based on the knowledge gained in the lecture. The lab situation also provides an opportunity for self-learning with motivated students adding extra complexity to given problems and generating their own program specifications. If there is a range of problem material available, ranging from easy to difficult, the better students may advance at their own pace. The underlying pedagogy of the lecture and lab scenario very much depends on how lecturers use the lecture and the lab. Lecturers could adopt a very rigid didactic approach in lectures and provide rigid problems for the lab sessions or they could adopt a very discursive, interactive collaborative problem-solving approach for the lecture supported by a guided self-learning approach for the labs where the students have access to scaffolding material and web resources to aid them in their knowledge construction.
Software Visualisation

The theory of Multiple Intelligences state that humans are born with a certain amount of intelligence, with specific intelligences being dominant and others recessive and the potential to develop all intelligences being possible. (Gardner, 1993). Spatial intelligence is the ability to form a mental model of a spatial world, object or pattern and then manoeuvring the image to operate in that environment. Research has shown that spatial abilities are very important in predicting programming success. (Scanlan et al, 1985).

What is Software Visualisation?

Software Visualisation is the practice of mapping abstract ideas represented in the code to visual representations that get the working s of the system across to the observer more easily (Ayrapetov & Graham, 2002). In other words software visualisation is used to aid the programmer/user of a program to understand the artefact being observed. Software Visualisation is a relatively young development with a lot of advances being made in the 1990s. Software visualisation environments are developed as programming tools for professionals or instructional tools for demonstrations or interactive study. According to Yehezhel (2002) “visualisations have cognitive potential which is effective if exploited by the learning environment to create a fructuous interaction between the student and the visualisation”. Ayrapetov & Graham (op. cit.) categorised software visualisation as follows:

- Program Visualisation
  The area of software visualisation that aims to help with understanding the programs code. This type of visualisation focuses on the graphical representation of an executing program and its data.
- Algorithm Animation
  An area mostly tested in instructional use and shows operations fundamental to an algorithm, as opposed to just code and data.
- Visual Programming
  The use of visual components to construct the program, usually with no textual representation behind the graphics ones.
• Programming by Demonstration
  These systems are more in the realm of artificial intelligence crossed with programming languages where programming is done by walking through the steps of an algorithm with the computer making sense of the steps.

• Computational Visualisation
  The area of visualisation that deals with viewing statistics as a diagram that allows certain aspects of the statistics to be clearly conveyed, e.g. viewing hot spots in the code in terms of error count.

Software Visualisation Taxonomies
A number of taxonomies exist which have focused on different characteristics of visualisation. Myers taxonomy for program visualisation classifies systems on two fronts, program aspect, code, data or algorithm, and display style, static or dynamic. Roman and Cox’s taxonomy is derived by viewing visualisations as a mapping from the program to a graphical representation whose characteristics are scope; that is what aspects of the program are visualised e.g. code, events, data; abstraction level, that is the level of the concepts displayed; specification method, that is the mechanism used by the animator to construct the animation, e.g. a fixed mapping from program to visualisation or by annotating the program with procedures to draw and modify images; user interaction, the viewer can have low interaction with the visualisation (passive) or frequently interact by selecting data or controlling speed for example (Yehezkel, 2002).

Motivation
Visualisation tools can help computer science instructors in a number of ways, ranging from catching the students attention to constructing in-depth exercises that engage the students. A number of motivating factors for instructional use of visualisation tools were presented by a report of the Working Group on Visualisation, chaired by Tom Naps (Naps et al 1996)
• Clarification of complex concepts through the use of pictures
• Alternative presentation mode
• Naps et al (op. cit.) reference research carried out that indicates that some students think better in visual terms
• The visualisation tools can be used as a hook by which lecturers can grab the students’ attention
• Visualisation tools allow lecturers to cover more material in less time
• Good visualisation can increase the students’ understanding
• Visualisation allows students to investigate the problem being studied, altering inputs predicting next steps and so forth thus encouraging active experimentation
• Motivates the students by allowing optional assignments to challenge the stronger students and allowing self-paced exploration
• Visualisation tools facilitates visual debugging thus aiding students in this often frustrating stage of development
• Visualisations save the lecturer time in manually illustrating structures in the classroom and increases the clarity of presentations
• Visualisation can be enhanced to incorporate other perceptual components such as audio
Program Visualisation
This type of visualisation focuses on the graphical representation of an executing program and its data. Data, code and events of interest are visualized; there is a low level of abstraction whereby there is direct representation of the code and data; each aspect of the program to be visualized is predefined and the role of the user can be from passive to active with predefined interactions such as changes of input data and speed (Naps et al 1996). A number of tools have been developed in this area.

JAWAA - Java and Web Based Algorithm Animation
In Duke University in Durham visual and interactive tools are used by instructors to teach computer science and by students to learn concepts in a visual way, one such tool is JAWAA. JAWAA is a scripting language that can be the output of a program and it runs over the Web with no installation. One can enter JAWAA commands in a file, modify a templated HTML file and generate an animation that runs on a web page. One can create primitives such as circles, rectangles, lines and text and move them around singly or in a group and with advanced features of the tool one can create stacks, queues, trees and graphs. There is a speed control bar and an ability to pause or restart the animation.

JAWAA is used in the CS1 course where the students learn programming using C++. Here JAWAA is used by hiding the details of JAWAA in classes. Students are given a prewritten C++ class and JAWAA is automatically output into a file that students can load onto a web page. For example, students learn about a hot air balloon class and control it by making it ascend, cruise and descend. In the early part of the course they can create the balloon and JAWAA output is automatically written to a file when they run their program. Later in the course they can create a hot air balloon race in which they there are several balloons that randomly move one to four steps each. Students can check the distances of each balloon to decide which balloon wins the race, thereby controlling and interacting with the tool.

Figure 1: JAWAA Hot Air Balloon Simulation
JAWAA is used in the CS2 course for animation of data structures. In that course the students write JAWAA commands as output in their programs, and with JAWAA students can create simple primitives and data structures easily. Students can decide which data structures to animate, enabling them to animate a data structure or concept that they may feel is difficult, thereby increasing their understanding. They are in control of how they use the tool.

Other courses use JAWAA in two ways, one to create animations of concepts and two to animate part of their program. As an example of program execution students can animate a CAMO (Cat and Mouse) program, which is a simple language in which you declare cat, mouse and mouse holes and move the cat and mice around. With JAWAA the students can see their program execute.

JAWAA and associated tools are available from Susan Rodgers’ web page

http://www.cs.duke.edu/~rodger/tools/tools.html

**EROSI – Explicit Representer of Subprogram Invocation**

Learning recursion and applying recursive principles to problem solving have been identified as a special problem area for students (George 2000a). George concludes from previous research that novices easily understand tail recursion but have difficulty in understanding embedded recursion where the recursive call is embedded into a group of algorithm statements, so there are other actions to be performed after the recursive call. This, he believes, is because the students do not have proper mental models of recursive processes, this problem being subsumed into their problem of not having an effective mental model of subprogram invocations. George argues that many novices see the subprogram as a “*static segment of program code above the main body which is executed*”. He argues for aiding novices in visualizing a more sophisticated subprogram execution model, the “*dynamic-logical model*”, where each invocation involves suspending the calling program, transferring execution control and sometimes data to a new copy of the subprogram, and then transferring control and data back to the calling program after the subprogram execution has ended. Each subprogram is visualized as physically separate from the original subprogram having its own data set. This model, argues George, provides an understanding of the process of recursion.
EROSI

George developed the EROSI tutor to aid novices in developing the “copies” model of recursion. “In the copies model, recursion is defined as a process which is capable of triggering new instantiations of itself with control passing forward to successive instantiations (active flow) and back to suspended ones (passive flow)” (George 2002).

The EROSI tutor uses visualisation and animation and sound and colour to simulate the dynamic-logical model of subprogram calls. It facilitates dynamic code visualisation by highlighting parts of the code being executed, and dynamic algorithm visualisation by generating an animation of the algorithm in action. The EROSI tutor allows the user to visualise

- The sequential execution of a program, subprogram
- The suspension of a calling program at the point which it invoked a subprogram
- The active flow of execution control going from the point where the subprogram is invoked to a new “copy” of the subprogram. Also passive flow of control from the copy of the subprogram back to the previously suspended calling program.
- Flow of data, each variable traced through the series of transformations in which it is involved
- Resulting actions/output of a program/subprogram
- Separate invocations of subprograms

The user interacts with the system by selecting a menu option, which allows him to vary the pitch of system sounds and the execution speeds of features such as “flow of control” movement and the invocation of subprogram copies. The tutor menu offers the following menu options, Subprograms without parameters, Subprograms with parameters, Complex calls and Recursion, all which contain sub-menus with programs whose execution can be animated. The order of the menu options reflect a learning hierarchy and programs increase in complexity, culminating in the Recursion option where learners can view simulations of various recursive subprograms entailing tail or embedded recursion.

To start a simulation the user is presented with a screen which contains the full text of the program to be executed, a window to show program output and instructions to start, to start the user may be required to enter a program variable or parameter. During the simulation, the lines of code, which are being, executed change colour and a beep is sounded. On arrival at a subprogram call an arrowhead appears at the right of the statement and moves away from the calling program leaving
a dotted trail to a position where a new window containing the subprogram text with updated variable values pops up. The dotted line reflects the transfer of control from the calling to the subprogram and the new window conveys the idea of a new “copy” of the called program. The user may pause and resume as required. After the execution of a subprogram, control is returned to the calling program using arrowheads. The execution history of the program remains on the screen after execution of the main program thereby allowing the user to review the process.

![Figure 2: Snapshot of tail recursion in EROSI (Source: George 2002)](image)

George concluded that the teaching of recursion using a visualisation approach should be accompanied by appropriate techniques for the construction of recursive programs as some subjects who seemed to have understood the model could not successfully construct a solution. However he also concluded that the use of a visualisation tool as his could provide support for construction tasks.

**Alice**

Alice ([www.alice.org](http://www.alice.org)) is a 3D interactive animation environment, that provides a graphic visualisation of a program’s state in an animated small world and thereby supports the novice programmer in learning to construct and debug programs (Dann, Cooper & Pausch 2000).

**Motivation**

Alice was built by the Stage 3 Research Group at Carnegie Mellon University under the direction of Randy Pausch. The goal of Alice was “to make it easy for novices to develop interesting 3D graphic animations” (Dann et al 2000a). The developers felt that Alice would aid students and
“actively engage” them in increasing their knowledge and skills in the areas of developing algorithms, figuring out how to apply problem solving techniques in their programs and using common programming constructs. Pausch and his team felt that students needed to develop an increased level of competency in how to design an algorithm for solving a problem and how to use programming statements in accomplishing that goal. They also observed that students were unable to visualize the steps of the execution of a program and therefore were not able to figure out what went wrong when things did not work. To this end they used Alice, an environment in which students can learn problem solving strategies and the necessary concepts and skills needed to create a computer program. Animation of program execution can be used to help the students tie all the pieces of a program together.

What is Alice?

Alice is a 3D interactive graphics programming environment for Windows. It is primarily a scripting and prototyping environment for 3D object behaviour. 3D models of objects (animals and vehicles) populate a virtual world in Alice. Alice has an object-oriented flavour, and by writing simple scripts users can control object behaviour and appearance. During the script execution, objects respond to user input via the mouse and keyboard. Alice is built on top of the programming language Python (www.python.org).

Alice has a set of actions that can be subdivided into those that tell an object to perform a motion and those that change the physical nature of an object, including moving objects and rotating objects, destroying an object, dynamic object creation and making an object visible and invisible. In Alice it is possible to name a sequence of instructions, e.g. making a bunny hop by a sequence of Move and Turn instructions enabling the students to immediately visualize the program constructs. Alice supports functions, which are mainly used in recursion, looping, implementation of interactions via events and computation; decisions, supported through the underlying Python language. Due to the visual feedback students can see immediately the results of a decision statement. Alice supports looping through the Loop instruction, and recursion is supported through the SetAlarm command and by using a visual representation of a recursive function called Chase, in which a fish moves towards a cat until it is within 2 distance units of the cat, Pausch and his team found that students found the recursive action easy to understand. Authoring in Alice consists of creating an opening scene and scripting, the scripting always starting its execution from the saved opening scene.
Conway et al (2000) documented findings from using Alice as empirical research results, they were supported by formal observations of 100 users and hundreds of informal observations over a 4 year period. The authors conducted tutorial observation sessions where the subjects had no programming experience and they were tested using a two-person talk-aloud protocol over a period of 30 minutes. The authors gathered observations from other sources, such as suggestions from 20 graduate students in a graduate-level graphics class and exchanging emails with members of the Alice community, among others. Where the findings were used as a feed into the design of the system, they are interesting from the programming point of view. The results show that “novices are strongly influenced by surface issues, and seemingly inconsequential name choices can often make the difference between a clear API and a confusing one”. Using names such as, Resize, not Scale, Move not Translate AsSeenBy not CoordSys, for example, was seen as quite powerful. Other observations about novices included

- Typing is hard- novices appreciated mouse control and dialogs
- Problems in 3D perception- a small percentage had problems with the depth of objects
- High Expectations- the novices expected collision detection and gravity.

Figure 3: Sample world from Alice
Becker (Becker 2001) and his colleagues in the University of Waterloo, Ontario, wanted to develop a Java-based introductory programming course but had difficulty in sourcing a satisfactory textbook which introduced objects early. They discovered Karel the Robot in *Karel++: A gentle introduction to the Art of Object-Oriented Programming* but as it was oriented towards C++ they got permission from the publishers to translate Karel++ to Java and wrote software to support Karel, then they had a course with which they were happy.

Karel’s world is very simple, there are avenues running north and south numbered 1 to infinity and streets running east and west, also numbered one to infinity. Walls may block avenues or streets and beepers may be placed on the intersections of the avenues and streets. Several robots may exist within the same world. These robots may move forward from one intersection to an adjacent intersection unless the way is blocked by a wall, they may turn, may pick up a beeper or place beepers, determine what direction they are facing or if there is another robot on the current intersection. The world is shown visually,

*Figure 4: An initial situation with one robot(arrowhead), beepers(circles) and walls(rectangles) (Source: Becker 2001)*

Becker’s implementation of Karel allows the full power of Java and when the students write a robot program they simply import the required classes from a library. The world is a fully extendable class and the students are encouraged to extend the world with new behaviours. Karel is used for the first 4-5 weeks of the course to teach object instantiation and extension, selection and iteration, methods with parameters and instance variables. Becker outlines the advantages of using this approach as follows

- Karel emphasises the fundamental concepts in OO programming from the beginning
• Karel allows the fundamentals of procedural programming to be introduced naturally with OO fundamentals
• Using a picture of the initial state of a problem and another of the final state makes it easier for students to understand the actual problem
• The animation provides visual feedback
• Students found the robots fun!

JKarelRobot
JKarelRobot is an educational software tool, developed by Buck and Stucki, (Buck & Stucki, 2001), which extends the concept of Karel the Robot with Bloom’s Taxonomy of Educational Objectives as a guiding principle, thereby supporting an Inside/Out pedagogy in an introductory programming course. This Inside/Out pedagogy involves an incremental, graduated exposure to complexity and structure based on the levels of cognitive development described by Bloom. The authors found that Karel the Robot has limitations in that it does not provide direct support for the Knowledge, Comprehension and Application levels of Bloom’s Taxonomy, as the students are forced into the Synthesis level by the software and it is only used by the students for a short time before the students outgrow it. Due to these limitations Buck and Stucki extended Karel in two ways, one to support more directly the primitive levels of cognitive development and two to teach more concepts and support more of the curriculum. JKarelRobot is written in Java and supports teaching with Pascal, Java or Lisp syntax.

The authors smoothly animated Karel’s moves and turns so that the students could see the robots’ actions as they were carried out, not simply the beginning and end state of each instruction. The authors also believed that to write a whole program was too large a step for students learning Karel so an interpretive mode was developed to JKarelRobot so that the students could type in one or more statements and see Karel carry them out immediately, if the statements are incorrect syntactically feedback is given to the students.
The authors found that the interactive environment provides graduated experiences through the cognitive levels. At the Knowledge level they have the statements and their syntax. At the Comprehension level is the capability of predicting what the statement will cause Karel to do, they provide support for this by asking the students to predict what’s the next statement to be executed as the students single step through their program. The students are given a score at the end of the run. At the Application level the student can apply a statement to achieve the desired effect. Flowcharts are used at the Analysis level by asking the students to translate a given program from language control structures into flowcharts, thereby reinforcing the meaning of the control structures, particularly nested controls structure.
Other Analysis exercises are the identification and correction of errors and the modification of a given program. At the Synthesis level they are asked to write a program that meets a specification. At the Evaluation level the students are asked to compare two alternative implementations that meet a common specification.

The authors claim that the difference between this approach and teaching with a traditional programming language is that the state of the world is more familiar to novice students than the concept of a variable so that they have less interference with learning the control structures. Buck and Stucki have added Boolean expressions and procedure parameters to JKarelRobot and plan to include recursion in the future. The authors believe that Bloom’s Taxonomy provides a basis for more efficacious pedagogy in particular in identifying topics, exercises and assignments. Web-based materials available at: http://math.otterbein.edu/JKarelRobot

AnimPascal

AnimPascal (Satratzemi et al, 2001) is a program animator that incorporates the ability to record problem-solving paths followed by students. The aim of AnimPascal is to help students understand the phases of developing, verifying, debugging and executing a program. The authors reference studies conducted on the comprehension difficulties students meet when they are taught introductory programming concepts, some are summarised as follows

- Difficulties attributed to the syntax and semantics of programming languages
- The need to understand the established programming structures
- The need to learn how to design, develop, verify and debug a program when given certain tools
- Lack of visualization of program execution makes understanding of language semantics hard

Based on the above findings the authors developed AnimPascal, its uniqueness being that it records the students actions which reflect the steps in the problem-solving paths followed by the students. It satisfies two goals a) to help novice programmers develop, verify, debug and execute their programs and b) to help teachers detect students misconceptions about programming.

AnimPascal facilitates the ability to edit and compile standard Pascal programs, the dynamic visualization of program execution, and the recording of different versions of user programs and associated compilation outputs. An option Animating allows the dynamic visualization of program
execution, the current source statement is highlighted and the result of it’s execution, affecting one of the following areas, Display Variables, Program Output, or Program Input is generated. Every time a student compiles his program the systems automatically records the new version of the source code and the corresponding compiler output then a History of Compilations option presents the common mistakes of the students. This can help teachers realise common mistakes and misconceptions of their students, and address them in lectures.

The authors recorded their findings of using AnimPascal in a first year laboratory where the students were asked to implement the binary search algorithm, which had been taught in class some time before. Satratzemi et al were able to generate a concise analysis of the paths to the solution followed by the students and this enabled them to find that:

- Given the repeated attempts of students to correct syntactic errors, novice students should be introduced to languages with simple syntactic rules and syntax can be a negative factor in problem solving.
- Students were able to proceed where AnimPascal provided hints as opposed to where the compiler generated error messages were produced which seemed to be of no aid to the students.
- The timeline generated from the results highlights the parts of the algorithm that the students find most difficult.

The authors claim that AnimPascal appears to be of great help to novice programmers in helping them improve their ability to design, develop verify and debug their programs and it has benefits in providing teachers with information regarding the technique used by the students. Where one would concur with the benefit of user-friendly error messaging enabling the students to debug their programs and the obvious benefit of providing teachers with the problem-solving path followed by the students, one would question the benefit of the tool in terms of aiding program design.

**Logo**

“Logo is the name for a philosophy of education and a continually evolving family of programming languages that aid in it’s realization” Harold Abelson, Apple Logo, 1982.

The Logo programming environments have been developed over the past 30 years are rooted in constructivist educational philosophy, and are designed to support constructive learning. The first version of Logo, a dialect of Lisp, was created in 1967. Logo was designed as a tool for learning.
Its features - modularity, extensibility, interactivity, and flexibility - follow from this goal. It is designed to be accessible to novices but also supports experienced users. The most popular Logo environments have involved a Turtle, originally a robotic creature that sat on the floor and could be directed to move around by typing commands at the computer. Soon the Turtle migrated to the computer graphics screen where it is used to draw shapes, designs, and pictures. In Logo environments with many such turtles, or "sprites" (different species) as they are sometimes called, elaborate animations and games can be created. Logo is easy to use as Figure 7 illustrates.

forward 60 moves the turtle forward 60 pixels, square is a simple procedure that draws a square

You could draw a flag:

```
forward 60
square
back 60
```

You could make a circle of squares:

```
repeat 12 [square right 30]
```

**Figure 7: Logo Examples**

Logo is structured similarly to Pascal, you create a main program which calls other procedures and functions, passing variables as needed. It is a recursive language and offers a full range of disk and file handling commands. Widespread use of Logo began in the 1970’s with the advent of personal computers. This prompted the generation of new versions of Logo for the Apple II and Texas Instruments TI 99/4. The Logo language was introduced to elementary schools and with the publication of Papert’s Mindstorms it became very popular with teachers. At the end of the 1980’s there was some interest in using Logo as a "serious" programming language, especially for the new Macintosh computer. MacLogo added new functionality to the Logo environment. Coral Software, developed an object-oriented version of Logo called Object Logo. It included a compiler.
which allowed programs to run at higher speed, and stand-alone applications could be created. But Logo did not become popular among applications programmers. There has been a lot of innovation within the Logo community, from LEGO Logo, LogoWriter, which included word processing capabilities to the Programmable Brick and Micro Worlds. MicroWorlds is a modern Macintosh application with an interface that is familiar to users of other Macintosh programs. MicroWorlds includes drawing tools, a shape editor, a melody maker, the ability to import graphics and sounds that work along with Logo to support the creation of multimedia projects, games, and simulations. MicroWorlds Logo includes a number of changes, the most significant being multi-tasking, or parallel processing. Multi-tasking has recently been implemented in PClogo for Windows, as well. StarLogo (Resnick 1996) is an extension of the Logo language designed to help non expert users model and explore decentralized systems based on parallelism. Studies (www.terrapinlogo.com) have shown that students who use Logo:

- plan more efficiently
- represent planning tasks differently
- have an increased understanding of geometry
- persevere in solving problems
- are better at resolving conflict
- are more self-directed
- exhibit desirable social interactions

However some studies have had mixed or inconclusive results.

Doug Clements summarizes Logo research results (Logo Exchange, January 1988):

*Logo's potential to develop geometric ideas will be fulfilled to the extent that teachers help shape their students' Logo experiences. Students do not automatically transfer knowledge gained in one situation to another. Repetition is not sufficient. Questions that cause students to reflect on what they were doing are instrumental*

The conclusion from the research being that the teacher’s role is critical to the students’ success.
Turtle graphics is a popular tool for teaching programming and Schaub(2000) uses the turtle to introduce basic programming concepts in a CS1 course. Java is the language being used for the course but Schaub uses a hypothetical turtle language to explain what an algorithm is, abstraction and its benefits, variables, and selection and looping constructs. When the students are ready to learn Java syntax, Schaub provides a Turtle class to translate their turtle algorithms into working Java code. The following code draws a T

```java
import cpsll0.turtle.*;
public class DrawT {
    public static void main(String[] args) {
        Turtle t = new Turtle( "Herb");
        t.pendown ( ) ;
        t. forward (30) ;
        t.right (90) ;
        t. forward (10) ;
        t.backward (20) ;
    }
}
```

Using the Turtle the students learn to read method interfaces so they can call methods properly and they learn to define methods. At this stage the students don’t know anything about AWT and have only a basic knowledge of what an object is. Schaub then moves the students on to Java AWT for the remainder of the course. However no evaluation of the use of the Turtle paradigm was conducted to indicate whether the students found it beneficial or whether it actually improved learning. The Turtle class is available at [http://www.bju.edu/cps/faculty/sschaub](http://www.bju.edu/cps/faculty/sschaub).

**Introducing Computer Science through Animation and Virtual Worlds(Rodger)**

At Duke University a course has been developed, Animation and Virtual Worlds, for non-majors to teach computer science concepts and programming through simple animation and 2D and 3D virtual worlds. By teaching programming through objects in a visual and animated way the students are motivated and the classes are taught in a room with computers where the students work in pairs and the total class size was 15. The lecturer gave a short lecture at the start of each lecture to introduce the topic and then the students work on the computers.

The students begin the course by learning HTML so that they can build a web portfolio. The concepts of algorithms and preciseness are introduced in this phase and the students develop
algorithms for simple tasks such as sandwich making. These algorithms were enacted in front of the class to illustrate that they were not precise. They also spend a day investigating sorting. In the next phase they learn to create simple animations using JAWAA, firstly creating several simple objects and moving and grouping them. Other exercises was for the students to develop an animation that had certain requirements, another to develop a traffic simulation and thirdly a sorting assignment.

In the next phase StarLogo(Resnick) was used. The students learned StarLogo through a tutorial and then created a traffic simulation that had similar requirements to the JAWAA traffic assignment. The students then created any StarLogo project, describing it with a web page.

Figure 8: StarLogo Traffic Simulation (Source: Rodger)

Alice (www.alice.org) was then used by the students to create a 3D world. The students spent one and a half weeks using Alice during which time they created a world with specific requirements: a flying logo, coordinated movement, interactive action, making a 3D object using the Teddy tool and including that object in the world. After completion the students put the world on a web page with a description of how to interact with the world. The last five weeks were spent using Karel++, a tool for programming robots that move graphically in 2D worlds. This is an object oriented version of Karel the Robot, and has a “substantial but gentle” programming component. The students worked on creating new types of robots, modifying inherited instructions, conditional instructions, nested conditionals, looping structures, solving a maze and recursion.
Three evaluations on the dynamics and topics of the course have been conducted with the following results:

- The majority of the students liked working in pairs
- Web page creation and Alice were the preferred topics with 100% of the students reporting they liked these units
- StarLogo and Karel++ were next in terms of popularity with 75% of the group liking these units
- JAWAA fared least well with only 50% of the group liking this unit – this tool is going to be used with a graphical editor to create animations rather than typing commands into a file, for the next offering of the course.

The authors acknowledge that while Alice was the most popular tool it is the less like programming and are going to include some python programming with Alice for the next offering of the course. Some students also felt that Karel++ was too complex.

While the evaluation determines whether the students liked the course it does not indicate whether running a course as this helped the students learn to program any better than a traditional approach. It also indicates that regardless of the interface once students are exposed to less interactive and complex programming concepts they find it difficult, e.g. Karel++. Some of the students on this course went on to enrol in the CS1 course and it would be interesting to see if they had any significant benefit over the other students who did not attend this animated course. However it does appear that by using visuals and animation that the student’s motivation has been increased.
Benefits of Program Visualisation

- The Visualisation engages student (Dann et al, op. cit.)
- Software tools that provide an animated and visual view are beneficial (Rodger op. cit.)
- Availability, ability to download a variety of tools from the web
- Students can select the concept to animate, it is not just a predefined example (JAWAA)
- Students can control and interact with the tool (JAWAA, EROSI, Alice)
- Diagrammatically traces recursive calls – useful for students (George, op. cit., Dann et al, op.cit.)
- Being able to change code and quickly see animated results maintain interest and make the program easy to debug (Dann et al, op. cit.)
- Students may view the 3D world as the state and look to see the location and orientation of objects in the world (Dann et al, op. cit.)
Algorithm Animation

Motivation
A great deal of time and resources have been devoted to developing animation systems for teaching Computer Science algorithms (Wilson & Aiken 1996). Algorithm animation shows operations fundamental to an algorithm, as opposed to just code and data. They are general-purpose tools, and the user decides what to visualize. The level of abstraction to be shown may be chosen. The user may define the relationship between the algorithm and the objects to be shown by means of annotations, i.e. the user chooses the images to be visualized and when to visualize them. The level of user interaction is defined by the programmers, it can range from passive user to a high level of interaction. (Naps et al 1996).

The first major interactive algorithm animation system was developed by Mark Brown in the early 1980s, BALSA (Brown University Algorithm Simulator and Animator). It was the prototype system for all software visualisation systems and it was used as an aid to algorithm design and analysis (Wiggins 1998).

Animation of sort algorithms goes back to 1981 when a video Sorting out Sorting was presented at SIGGRAPH. It showed views of data being sorted by different algorithms to help students understand how they work and how they compared to each other. Since then there have been many algorithm animation systems developed, such as BALSA, as above and GAIGS (Generalized Algorithm Illustration through Graphical Software)(Naps 1990).

SAMBA
John Stasko carried out research to empirically evaluate the pedagogical value of students viewing and interacting with prepared algorithm animations (Stasko et al, 1996). The results of the experiments carried out suggested that the benefits of animations were not obvious. These results and the fact that algorithm animation had not reached the use and application in instructional settings as hoped for motivated Stasko and his colleagues to seek alternative ways of utilizing algorithm animations in instructional settings. This prompted the question

“What if the students built the animation of algorithms themselves as opposed to simply interacting with algorithms already prepared for them?”
The developers felt that an algorithm animation system designed to foster student construction of animation involves certain requirements

- The animation systems must be easy to learn and to use
- Development of the animation must be intimately tied to the algorithm and its operations.

The Samba algorithm animation tool was created to meet those two requirements. Samba is an interactive animation interpreter that reads ascii commands and performs the corresponding animation actions. One set of commands create graphical objects for the animations e.g. an example of a command is as follows,

line 13 0.1 0.1 0.2 0.2 green thin
circle 27 0.8 0.7 0.1 red half

These commands create a line and a circle respectively. Each command has a number of different fields, the first is the command type and it defines the number of trailing fields in the command. The second field, 13, 27 as per example, specifies the unique string identifier for the object being created and the trailing fields specify the visual aspects of the object, e.g. the circle command provides a centre x and y co-ordinate, radius, colour and fill style in order. Other command are available to modify objects e.g.

color 27 blue

Object movements can be discrete jumps to new position or smooth actions. Commands exist to alter the viewing windows, zooming and multiple windows can be used. Groups of commands can be ‘batched’ so that they run concurrently.

![Figure 10: Student created quicksort animation (Source: Stasko 1996)](image-url)
Samba is a front end to the POLKA algorithm animation system. POLKA is a general purpose animation system, descended from the XTANGO system, but it is more powerful and flexible. According to the developers Samba is a “simpler more accessible front-end to POLKA”. The original version of the system is developed in C++ on top of UNIX and X Windows system but a newer version of the system (Samba and POLKA) is available which runs directly on top of Windows. Information on the system and both versions are available via anonymous ftp from ftp.cc.gatech.edu on the web site http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html

**Developing Samba Animations**

A student must annotate the implementation of an algorithm with ‘print’ statements to generate the commands to drive Samba. When the program executes these print statements will be output and comprise a trace of the programs operations. This output trace is then forwarded to Samba, which generates the specified animation. The transmission of commands can be interactive, with the output piped to Samba e.g.

% shellsort | samba

Alternatively the shellsort program can store its output in a file as follows

% shellsort > tracer

and then that file can be given to Samba as required e.g.

% samba tracer

Samba allows users to step 1 command at a time, if required or pause at any point. A key benefit, as defined by the developers, is its use of simple ascii commands as input. No new methodology is required and an algorithm written in any language can be animated, as all that is required is to output the ascii commands.
The Sort Animator

This is an algorithm animation tool that has been developed by Dershem & Brummund (1998). The advent of the WWW and Java applets opened up a new era for algorithm animation allowing remote interactive access with platform independence and the ability to link animations to text, sound and video. Algorithm animation can be coupled with code animations by including a separate window or frame where the code is displayed; the line of code being executed is highlighted simultaneously with the algorithm animation activity. Dershem & Brummund referenced systems where this had been implemented, e.g. JCAT and Zstep. The research carried out by Stasko (1997) indicating the pedagogical benefits of user-designed animations was taken on board by Dershem & Brummund. Stasko’s research coupled with the advances in web access and code coupling lead to Dershem & Brummund developing a tool for teaching and learning sort algorithms. They also added an animation technique to enhance student learning of how recursion occurs in sorting algorithms.

The Sort Animator is a Java applet that provides a split window showing the sort animation in the left frame and the code animation in the right frame. Vertical bars are used to represent the elements to be sorted while horizontal co-ordinate showing the index of the element in the sort list, the vertical co-ordinate representing the magnitude of the elements’ sort key. The bars move as the algorithm executes and as data movement occurs in the sort list.

A number of controls exist for the user, all of which can be modified at any point in the animation:

- Colour, for background, bar and to highlight the lines of text.
- User may specify the number of items to be sorted, up to 900.
- The arrangement of data can be random, ascending or ascending.
- Speed control to manage the algorithm execution, by use of a slider.
- User may pause the animation as required.
- Explanation button, when selected invokes a window with a text-based explanation of the sort algorithm
- Algorithm button that allows a selection of 1 to 6 algorithms.
- The number of comparisons and swaps made.
Figure 11: Sort Animator View (Source: Dersham & Brammond)

To animate a recursive sort algorithm, a number of horizontal bars is added at the top of the left frame to indicate the level of recursion and the part of the sort list that is sorted by each recursive call.

The Sort Animation Builder has the same animation environment as the Animator. The sort algorithm is provided by the user along with special instructions to direct the animation.

JHAVÉ  Java-hosted Algorithm Visualisation Environment

JHAVÉ is a client-server environment for delivering algorithm visualisation over the Web. JHAVÉ was designed and developed by Tom Naps, James Eagan and Laura Norton taking into account pedagogical issues (Naps, Eagan & Norton, 2000). Naps and his colleagues referenced research carried out into the effectiveness of algorithm visualisation systems in teaching algorithms and data structures finding that

- Passive modes of engagement does not significantly improve the performance of learning.
- Effect of visualisation may be lost unless it is accompanied by teacher-provided explanations.
- Students frequently get confused if no rewind facility provided with the animation.

Other research findings included

- Better results were achieved by forcing the students to be more “active”, e.g. construct own input data.
- By making predictions on what they would see during the visualisation students performance was improved significantly over just watching the visualisation (Stasko et al 1996).
These findings lead Naps and his team to conclude that to make the algorithm visualisation an effective instructional tool the students should be actively engaged by using hooks that force their interaction with the visualisation. The design goals for JHAVÉ were developed,

- Present students with at least two types of visualisations, smoothly running and sequence of discrete snapshots, both with rewind facilities.
- Supplement the visualisations with context-sensitive textual material in a web browser window.
- Provide students with input generators.
- Force the students into active participation by interrupting the visualisation with “stop-and-think” questions, making the students predict what they will see in the next step of the visualisation.

**JHAVÉ Architecture**

JHAVÉ is a client-server architecture, implemented in Java into which more specific algorithm visualisation engines may be plugged. Currently there are two such engines, one for the Samba animation scripting language (see Samba) and one for GAIGS language developed by Naps. Once such an engine “plugs into” JHAVÉ, designers using that engine can incorporate the JHAVÉ tools. Context-sensitive documentation, input generators and stop-and-think questions. These engines must produce a script file, which is parsed and rendered by the engine. In JHAVÉ the server application manages the available algorithms and generates the visualisation script files that the client can display. A session works as follows:

A web server launches an instance of the AVClient applet, which displays a list of available algorithms. The user selects an algorithm from the list, the client applet sends a request to server, which will run the program that generates the script file and sends it back to the client. Client renders it with the appropriate engine. If algorithms require input, server sends input generator object to client. Once the user fills in input data, the client sends it back to the server to be used as the data set to run the algorithm.
JHAVÉ and Animal

Rosling and Naps (2002) note that even though algorithm visualizations are numerous they have not been successfully adapted into mainstream computer science education and they claim that “algorithm visualizations need to better address pedagogical requirements for effective educational use”. They cite research in listing the following pedagogical requirements in an AV system:

- Reliably reaching a large target audience- Java applet, or application preferably
- General purpose systems- offer common interface to multiple animations I
- Allow users to provide input to the algorithm
- Rewind capability
- Interactive prediction- stop-and-think questions
- Integration with database for course management reasons, taking research into account where the interactive predictions showed no benefit, by providing a “quiz-for-real” mode where the student’s responses are stored in a database improves the effectiveness of the AV
- Hypertext explanations of the visual display
- Smooth motion, but offer the option of viewing the animation on discrete steps

The authors have combined two systems, JHAVÉ and Animal (http://www.informatik.uni-siegen.de/~inf/Software/Animal/index.html) to generate a prototype that satisfies these requirements.

Animal was developed by Rosling and Freisleben at the University of Siegen, Germany, as an answer to their needs for a new animation tool. It stands for “A New Interactive Modeller for Animations in Lectures”. It is written in Java and uses Java’s Swing library. Animations can be generated and edited visually on a drawing pane or by using a built-in scripting language or animation API.

The system uses JHAVÉ as the client/server component in which the animations to be viewed are selected, JHAVÉ then produces a script for that algorithm and the animation is displayed using the Animal Visuaizer. Animal provides smooth or discrete steps, and all controls in a forward or reverse mode.

The authors conclude that the emphasis is now changing in terms of “how students uses AV technology has a much greater impact than what AV technology shows them”. They claim that AV
systems must be painlessly incorporated into courses and then empirically evaluated and that JHAVÉ and Animal is a first step in this direction. They provide the following comparative analysis of existing AV systems.

**Figure 12: Pedagogical requirements addressed in AV systems**

*(Source: Roßling and Naps)*

1 = general-purpose system, 2 = input generators, 3 = rewind capability, 4 = structural view, 5 = interactive prediction, 6 = databases support, 7 = inclusion of hypertext, 8 = smooth motion. (X) indicates partial support.

<table>
<thead>
<tr>
<th>System</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAWAA</td>
<td>(X)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(X)</td>
<td>(X)</td>
</tr>
<tr>
<td>Jumbo</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(X)</td>
<td>(X)</td>
</tr>
<tr>
<td>IDS</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>(X)</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GAIGS</td>
<td>(X)</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ANIMAL</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JHAVÉ + ANIMAL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

The above authors, Roßling and Naps have further refined their integration of JHAVE and Animal to move towards Intelligent Tutoring in algorithm visualization (Roßling and Naps 2002a). They claim that most AV systems are too impersonal in their interaction with the learner and so do not provide as much motivation as may be possible. To move towards an intelligent system the previously defined pedagogical requirements have to be met but also added functionality has to be added as follows:

- Input Data Specification—depending on the algorithm being visualized the user should be able to enter all, or some of the input data and be able to see partially grown structures and react to them by inserting data of particular interest

- Visualization Display, navigation and control – provide smooth reverse visualization of effects to enable the user to be switch from current to previous states for enhanced learning; also learners will be able to adapt the display to their current environment from notebook users to high beam users

- Interactive prediction – feedback should be incorporated into the questions to help the learner understand the topic better, in the form of links to additional material or hints on what the learner may have understood incorrectly for incorrect answers; the questions have to be grouped based on topic area and once a learner has shown his understanding in this topic they may be dropped to prevent boring students; points may be assigned to questions and learners informed of their progress

- External Documentation—the inclusion of *pedagogically dynamic* documentation that is sensitive to the algorithm state and values being used but also to the degree of expertise
being demonstrated by the student i.e. more detailed documentation is presented for the learner having difficulty

- Reusable visualisation modules – the generation of visualization for a given topic is easier if it can be structured into reusable modules e.g. sorting visualizations can be structured into introduction to the algorithm, data acquisition, initialisation, actual sorting, efficiency and a statement of time complexity, support for reusable modules makes visualization generation easier for AV authors.

To realize the stated goals, the authors are using JHAVE and Animal because they have already are expressive in addressing the pedagogical requirements list previously and previous work they carried out showed that Animal as a front end for JHAVE supports extensibility, Animal incorporates the extensible scripting language AnimalScript which can be used as a base point for implementing the intelligent tutoring features above. The work was due to be completed in the summer of 2002 i.e. incorporating the intelligent tutoring features into the prototype after which the authors plan to analyse the benefits of the system.

This development moves the AV arena closer to the learner in a pedagogical sense and one looks with interest to the finding of the research. One has to bear in mind however how this prototype will be used in the learning environment, as a core learning tool or as an occasional teaching aid.

Ethnographic studies of a social constructivist approach to integrating algorithm visualisation technology into an undergraduate algorithms course were carried out by Hundhausen(2000). Hundhausen conducted studies in a junior-level algorithm course that included AV construction and presentation assignments, the students constructed their own visualisations and then presented their visualisations to their instructors and peers for discussion, with a view to, among others, answering the question “do the assignments promote learning – that is do they help students to participate more fully”. Field techniques used in the study stemmed from HundHausen being both a student observer and a teaching assistant, the techniques being participant observation, informal interviewing, semi-structured interviewing, questionnaires, field notes, audio and video taping of the presentations, collection and analysis of artefacts (storyboards and animations) and collection and analysis of student diaries, in which the students indicated what they did for the animation, problems encountered and the time they spent on the assignments. The two studies were based on
two separate offerings of an algorithm course that were taught by the same instructor during consecutive quarters. Hundhausen argues that on the Social Constructivist view, AV “holds promise as a learning tool through it’s ability to facilitate students access to two expert activities: AV construction and presentation”. In Study I the construction of the algorithm visualisations followed Stasko’s( 1997) recommendations and they were then required to present the animation to the instructor and classmates. The students used Samba to construct and present their animations, of which there were three. In Study II only one assignment was retained, in which students were asked to approach it in two phases, “animation prospectus”, in which the students used low tech materials, transparencies, pens, construction paper, scissors) to construct “visualization storyboards” of their animations, present these storyboards for feedback and then to use Samba to implement the storyboards, bearing in mind the feedback received from the initial presentation.

Hundhausen argues that according to Social Constructivism, gaining fuller membership within a community entails participating in the community in increasing expert ways, in this regard he argues that the AV construction and presentation exercises, as used in this course, gave the students an opportunity to engage in activities that were typically performed only by course instructors. He argues that the exercises motivated the students to act in more expert ways and they were motivated by the challenge and that their “competence appeared to be transformed”. His most significant finding was that, within the context of AV construction and presentation exercises, conventional AV software could distract students from concentrating on activities and concepts of the algorithm course. When supported by a “low-tech” version of AV technology, the AV construction and presentation appeared to focus students on relevant activities and concepts and enable them to participate more extensively, thus contributing to their learning. Hundhausen acknowledges that these are preliminary finding and plans to conduct more rigorous empirical studies to test these findings.
Benefits of Algorithm Animations

- The animations engage and motivate students (Stasko et al., op. cit.)
- Usage of tools helped students learn and understand the algorithms (Stasko et al., op. cit.)
- Students working with some tools, e.g. Samba, used it as form of visual debugger providing them with feedback (Stasko et al., op. cit.)
- Accessibility, tools available over the Web (Sort Animator and Builder)
- Tools that allow student construction of algorithm animation fosters learning and understanding (Stasko et al., op. cit., Dersham & Brummond, op. cit.)
- Some tools allow comparison of algorithms (Dersham & Brummond op. cit.)
- Visual animation that is supported by textual explanation increases understanding (Dersham & Brummond op. cit., Stasko, J., 1997, Naps et al., 2000)
- Tools that use “stop and think” questions foster deeper understanding, by forcing the students using them to actively participate with the visualisation (Naps et al., op. cit.), this has been questioned by Jarc but further research by Grissom & Naps indicate that if the questions are part of a “quiz for real” they do improve performance.
Software Visualisation – Pedagogy

The use of pictures and animations as educational aids is accepted practice. Visualisation is used to clarify complex concepts and to enable students to develop mental models of the underlying concept, an algorithm or the steps of the execution of a program. Students have different learning styles and some students think best in visual terms, (Gardner, op. cit., Felder-Silverman). “Constructivism claims each individual creates cognitive structures (models) when learning, sensory data is combined with existing knowledge to create new cognitive structures, which in turn are the basis for further construction, knowledge is also created cognitively by reflecting on existing knowledge” (Ben-Ari, 2001). Visualisation aids knowledge construction by developing viable mental models of complex concepts. The aim of a successful visualisation is to provide a tool that the students can control, that actively engages the student and that gives immediate feedback (Wilson et al op. cit., George op. cit.). These aims concur with the principles of constructivism, which states that knowledge is “actively constructed by the student, not passively absorbed from textbooks and lectures” (Ben-Ari, op. cit.). Constructivism states that cognitive operations, such as reflection and exploration should be encouraged and we see Naps (Naps et al 2000) incorporating these principles into his tool by means of rewind facility, stop and think questions and user input.

Smith and Webb (op. cit.) cite Mayer when he states that “meaningful learning” occurs when new knowledge is actively associated with pre-existing knowledge structures. If meaningful learning occurs then the learner will have understood the new knowledge. He argues that the provision of appropriate metaphors is a useful way to help students understand the new material. One such way is through the use of software visualisation.
Visual Programming

Visual Programming is categorised by Ayrapatov & Graham (op. cit.) as the use of visual components to construct the program, usually with no textual representation behind the graphics ones. Myers(86) states that “visual programming (VP) refers to any system that allows the user to specify a program in a two dimensional fashion”. There are a number of visual programming tools available for teaching programming.

Visual Basic

Visual Basic is currently the most popular programming language in the United States but it is not that common in Computer Science curricula (Hummel & Mehta 2002). The reasons for this, they claim may be that it is more object-based and not truly object-oriented or that it is available only on the Windows platform. Hummel & Mehta argue that while VB should not replace the use of more traditional languages such as C, C++, Java it does fulfil an important role in modern CS curricula.

Hummel who uses VB in a Programming Languages course, found that students readily adapt to its language and environment in a few hours of instruction, they find it easy to use and that students who previously struggled with programming concepts were excited and successful with VB, leading him to conclude that visual learners find VB much more supportive.

VB was introduced as a prerequisite for CS1 in Saint Xavier University where they found that many students had difficulty in CS1 where they were learning C++. This changed after they had the prerequisite of introduction to programming using VB and he found that VB is very useful in introducing students to programming. Miles (Hummel & Mehta, op. cit.) at Lenoir-Rhyne College found VB very useful as a prototyping tool as part of a Senior Project course where the students implement the design of a software product. She found that VB supports the easy creation of the user interface and this allows the students to evaluate customer interaction with the program and for early customer feedback. The students may not use VB to implement the product but it is deemed to be a very useful prototyping tool.

Harris ( 2001) attempts to study the dichotomy between the acceptance of VB in the programming community and the rejection of VB in the academic community and argues that it should be considered for CS1 and CS2 classes. He argues that more new applications have been written in VB over the past 5 years than any other language. The two market leaders are VB and Java and
45% of managers in a survey of 107 application development managers identified either Java or VB as one of the two languages most important for their future development needs. VB holds the market share, 25.6% (projected for 2003), of computer languages based on licenses sold. VB is the market leader but its popularity is not reflected in academia. In a survey of 20 colleges and universities chosen randomly from a list of 1529 colleges and universities, Harris lists the following languages used in CS1.

<table>
<thead>
<tr>
<th>Language</th>
<th>Number of college/universities</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>12</td>
<td>60%</td>
</tr>
<tr>
<td>Java</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>Pascal</td>
<td>2</td>
<td>10%</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>5%</td>
</tr>
</tbody>
</table>

*Table 4: Survey of languages used in CS1 (Source, Harris 2001)*

Harris lists the lack of resources available, e.g. data structures books in VB, in VB as a possible reason why VB is not the main CS1 language. He puts forward the availability of Java as an alternative as another reason why VB is not used more frequently. He argues that while the demand for Java programmers is being met, the demand for VB programmers is not. He also claims that VB provides an excellent way to implement Windows API function calls. He disclaims most of the technical arguments against using VB as unjustified and he argues that even though VB uses object references instead of pointers linked structures can be created using class modules. He also argues that most of the features of C++ are implemented in VB including friend functions, templates, static data members and multiple inheritance (through multiple interfaces). Inheritance can be emulated in VB6 through subclassing. However Harris does not see the situation changing much in the near future mainly due to the Microsoft label that is VB.

Stagecast Creator

Since the late 1960’s program language designers have been trying to develop approaches to programming that succeeds with novices. None has gained widespread acceptance. Smith et al (2000) have developed a new approach that eliminates traditional programming languages in favour of a combination of two technologies, programming by demonstration (PBD) and visual
before-after rules. A commercial product, Stagecast Creator, based on PBD, was introduced in March 1999, “enabling even children to create interactive stories and simulations.”

Stagecast Creator is a novice programming system for constructing simulations, the result of a research and development project, initially called KidSim. The goal was to make computers more useful in education and the co-inventors, Smith and Cypher, focused on simulations in order to empower end users, students (children) and teachers to construct and modify simulations through programming. The author’s first conclusion was that no programming language would ever be widely accepted by end users due to the difficulty of learning involved. To get over the language difficulty they used programming by demonstration (PBD). In PBD users demonstrate algorithms to the computer by operating the computer’s interface just as they would if they weren’t programming. The computer records the user’s actions and can then re-execute them later on different inputs. Creator does not represent each step in a program, rather the beginning and end states. Creator has a syntax, the parts of a rule, the lists of rules in an object and the list of tests and actions in a rule, but people can program for a long time before being aware of it. Figure 13, Defining a Rule by Demonstration shows the interactive, visual process of creating a rule.

![Diagram showing the process of defining a rule](image)

*Figure 13: Defining a Rule by Demonstration, (Source: Smith & Cypher)*

The rule may be read as follows

*If the engine is on a piece of straight track and there is straight track to its right*

*Then move the engine to the right*

Other operations such as Create, Delete, Set Variable operations can be recorded by demonstration. The Creator authors claim that it bases itself on involving all of Bruner’s three mentalities, enactive, iconic and symbolic, in programming. Enactive mentality results from PBD
when users manipulate images directly, drag and drop functions are enactive. Iconic mentality results from the before- after rules and symbolic mentality results from the use of variables.

Smith and Cypher have conducted hundred of hours of direct tests on children and adults over 5 years, most on children aged 6-12 in school settings. The longitudinal studies were carried out in several elementary schools in California where teachers integrated computer prototypes of Creator into year long curricula designed to improve their students problem-solving skills. Independent researchers at several universities in the U.S. and England conducted formal user studies of the prototypes, and these all answered affirmatively to the following two questions:

- Can kids program with the technology?
- Do they enjoy it?

The authors also found that where students learned Creator first and then a traditional language, they are better programmers than those who go directly into a traditional language.

The authors concluded that the early evidence suggests that this approach is more acceptable to novice programmers than traditional approaches. The domain is limited to visual simulations, which they claim brings the system closer to the user and Creator shifts the language design emphasis from computer science to human factors, thus making it more meaningful to users and easier to learn and use.

**SIVIL - SImple VIisual Language** (Masterson & Mayer, 2001)

SIVIL is a new visual programming language in development at Canisius College, Buffalo, NY, to “*teach novice programmers to think more deeply about programming*“. The motivation behind SIVIL was to determine whether a language that presents code in a non-textual way makes programming easier for novice programming students. It uses pictures and icons to symbolize all of its programming structures. Students can create chunks of code by connecting these pictures together and the code can be compiled, tested, debugged and viewed in a graphically rich environment. The authors don’t advocate the end of textual coding, they see SIVIL as an aid to be used with a textual language such as Java to help novice students better understand programming concepts. They see SIVIL’s greatest advantage being its appeal to visual learners, for example they use the a symbolic representation of a loop as a circle and boxes and arrows provide the paradigm of looping or information flow, there are no separate control structures to learn. They argue that even if SIVIL is only used as a lead into a textual language the students might better
understand the concepts of a loop than those who went straight into textual programming. The following gives an example of a guessing game in SIVIL where the user successfully hones in on a number that the computer secretly chooses. The authors claim the student forms a mental picture of a program where the central loop combines with a decision to send control into one of three directions. They argue that a visual pattern such as this is easier to remember than a text based programming pattern.

![Figure 14: A numeric guessing game in SIVIL (Source, Masterson & Mayer 2001)](image)

The authors claim that visual systems need not be cluttered with detail, leaving the visual representation of the main algorithm clear for the student. SIVIL has an ability to allow the teacher or student to specify their own set of pictures to represent structures in the language. This allows visual connections to pieces of code that relate to the users experience, it allows programmers to create a set of images that are recognizable by another reader, e.g. manager, and by using images that are familiar it enables better communication between engineers and management in the business world. SIVIL allows for documentation in a number of ways, they can be included as textual descriptions next to their encloser, these comments can be hidden or moved at will to any point of the SIVIL drawing board. The system also allows long detailed descriptions to be created
and hidden which allows additional information to be obtained if needed. There is also a higher level documentation space for entire programs. SIVIL has features that make the language appealing to advanced programmers, it allows primitives to be selected from a toolbox and add them to the coding window, it allows users open up enclosers and see the code inside it. It provide a debugging environment where the programmer is allowed to open up the coding window, view the program as it runs, inspecting the values received at each encloser.

Masterson an Meyer propose to use both SIVIL and a simple textual programming language in a CS0 course in order to improve students’ conceptual development. They intend to test the use of this system and plan to develop either a classical control experiment or a protocol experiment comparing programming in SIVIL to programming in a textual language, students might be asked to write SIVIL programs after learning a few basic patterns, another option they are considering is asking students to translate from SIVIL into Java as a means of using visual concepts to reinforce traditional programming. They based their system on Bloom’s Taxonomy and argue that SIVIL “seems to have the characteristics to succeed in aiding students in their pursuit of programming”.

The authors acknowledge that SIVIL has drawbacks, such as the problems to be encountered from going from SIVIL to a textual language such as C++ or Java but they feel if used correctly that it may be useful and meaningful. One looks forward with interest to the evaluation results.
Robots

Motivation

Researchers from Maria Montessori, who found that children learn through investigation and exploration, to Piaget who argues that knowledge is constructed through concrete operations, to Papert who calls for a “re-evaluation of concrete” as he suggests that abstract reasoning should not be viewed as superior to concrete manipulations, all promote learning through physical activity, the concrete before the abstract. This line of thought is being adopted by many researchers in the area of teaching programming and implemented by using robots to teach. This started as far back as the 1960’s with Karel the Robot (see Program Visualisation) and has now advanced into the realm of affordable physical robots.

“Research has shown that active learning, learning promoted by interacting with one’s environment, as opposed to lectures, is most effective in developing a students ability to acquire knowledge” (Linder et al 2001). Linder and his colleagues argue that active learning can be facilitated by using mobile robots in a collaborative setting. The use of robots as a metaphor for assisting students in understanding problem-solving in general is not new (Goldweber et al 2001). Simulated robot environments have been in existence for some time, particularly Karel the Robot, which is now available as an object oriented version Karel++, and has been used with some success in teaching Java programming (Becker 2001). In the late 1960s Logo was developed as a programming language for children. The most popular use involved a mechanical robot, a “floor turtle”, and with the proliferation of personal computers in the 1970s the focus shifted to “screen turtles”, which were much faster and more accurate than “floor turtles” (Resnick et al 1996).

LEGO/Logo linked the popular LEGO construction kit with the Logo programming language, children could build machines out of LEGO pieces using gears, motors and sensors and connect their machines to a computer and use Logo programs to control the machines. With the development of the Programmable Brick (Resnick et al op. cit.), a tiny portable computer embedded inside a LEGO brick, the robots, not only turtles now, were brought back into the real world.

So it is only recently that technology for inexpensively supplying real robots to undergraduates has become available, such as LEGO® MINDSTORMSTM (LEGO), MIT Handyboards and the Rug Warrior (Goldweber et al, op. cit.) which are programmable in languages such as C, Ada and Java and therefore allow these systems to be used for the purposes of teaching programming.
MINDSTORMS
The MINDSTORMS kit (LEGO), the commercial version of the MIT Programmable brick, is popular as a teaching tool, used in US Air Force Academy, Xavier University, University of Evansville, Villanova University (Goldweber et al, op. cit.) and many more sites. Fagin and his colleagues (2001) at the US Air Force Academy have developed an interface to the Lego Mindstorm RCs based on a subset of Ada, known as Ada/Mindstorms 2.0, as they felt that the original programming environment was not suitable for an introductory programming course due to “the unsophisticated error handling, confusing naming conventions and a failure to abstract away technical issues that could provide unnecessary stumbling blocks to students in an introductory computer science course”.
They used Ada/Mindstorms 2.0 as part of an experiment in determining the effectiveness of using robots to teach computer science in sections of their core computing course.
They began by teaching sequential control flow, variables and constants and used the robots to demonstrate each of the concepts taught, with variables they made the robot change its behaviour in a way directly related to the variable value, as code snippet indicates

```plaintext
--an integer variable
Time_Forward : Integer := 500;
Output_On_For(Output => Output_A,  
   Hundreds_Of_A_Second => Time_Forward);  
Output_On_For(Output => Output_C,  
   Hundreds_Of_A_Second => Time_Forward);  
Time_Forward := Time_Forward*3/4;

--now the robot goes forward for % as long
Output_On_For(Output => Output_A,  
   Hundreds_Of_A_Second => Time_Forward);  
Output_On_For(Output => Output_C,  
   Hundreds_Of_A_Second =>  
   Time_Forward);  
```

They then went on to teach procedures, selection, Boolean expressions and arrays in a similar fashion using the robots to demonstrate each of the topics taught finding that the use of robots made teaching procedures very natural. Selection was demonstrated by the robot’s ability to react to its environment via input sensors, the robot back up when it hit a wall and go elsewhere. The authors added support for arrays and had the students capture a sequence of numbers input to the robot through a combination of touch sensor and bumper presses. Once the sequence was captured,
the program “played back” the sequence by examining each number in order and had the robot do a predefined manoeuvre based on the number read. They authors taught 180 students using Ada/Mindstorms while about 800 in the same class year learned programming traditionally. Both sets of students were tested with written exams and individual effort, timed, non-robotics programming assignments. The results are awaited.

**US Military Academy, West Point**
The US Military Academy run a semester course on IT for all first year cadets, a large portion of which is devoted to problem-solving using computer programming(Schumacher et al, 2001). The course uses Java and as they place heavy emphasis on active learning, they introduced the LEGO Mindstorms robots. The faculty developed a computer simulation of the robot environment as well as a Java programming language translator for the RCX. These two tools are combined into a programming and teaching environment called Jago. Jago is a subset of Java with a number of pre-defined methods that make programming the robot easier. Jago allows programs to be run in a virtual environment or be converted to NQC(Not Quite C) and compiled for execution by the RCX. This allows the students to create and test their programs without access to a robot in a virtual world after which they may carry out final testing and evaluation on a robot in the physical world. The reasons behind providing the Robot simulator was to minimise the logistics and overheads associated with issuing and tracking robot kits for 500 students and achieve a learning objective. By using the simulator the students can design, code and test their programs and once confident in their algorithm they cross-compile it to NQC and download it to the RCX. The students demonstrate their work on a robot as part of a project presentation. Results from 6 projects given to student to use the robots were very satisfactory in that the students all met the minimum requirements and also excelled in the projects. The students that completed the robot projects stated a better appreciation for the design process and future work is to integrate the use of robots into homework assignments and fundamental programming concept demonstrations. More work needs to be carried out to gather empirical evidence as to the educational benefit and effectiveness of using robots over a traditional approach to teach programming.
MINDSTORMS has been used in the University of Canterbury, Kent, U.K. in an introductory programming course using Java. (Barnes 2002). The kits allow imaginative physical models to be built, some which may be programmed via the RCX™ processor integrated into them and the availability of bytecode–compatible replacement firmware for the RCX (leJOS) made it attractive for Barnes to use the kits to teach Java. Barnes argues that the physical models are attractive as they provide “tangible feedback to the students on the workings of their programs” and that “good design and planning have to be a high priority” for the students due to the “physical constraints of the coding and debugging cycle”. Barnes found that he had some issues with the use of the robots as a tool for introductory programming, i.e. the use of more advanced concepts had been introduced earlier than he would have liked, for example to implement event listeners; the difficulty for introductory students to link the control flow within a driving program and the sequence of actions that a physical model goes through, i.e. the concurrent event driven nature of the physical model. Barnes is also concerned that good object-oriented programming style could be distorted by the memory limitations of the RCX. The availability of a single small LCD line on the RCX means that lack of feedback when things go wrong can be a problem for students. Due to the inexact nature of a physical model’s movements some actions may be hard to configure, e.g. turning at a fixed angle. Despite this Barnes concludes that custom APIs could be developed that better suit the pedagogic needs of a course and more importantly he argues that it is better to use these models to “enhance and support an introductory programming course rather than as the basis for a whole course”.

Wolz (2001) acknowledged the fact that due to the immediate feedback given by many Integrated Development Environments (IDEs), many students depend on this technology to complete their development task and do not spend enough time planning and designing their work. With this as motivation she designed a laboratory experiment for CS1 students, “Robot Planning and Design”, in which students would develop an appreciation for planning and it would give the students an appreciation for “thinking first”, rather than mindless re-testing. This experiment was incorporated into a Java course. As there were only 2 LEGO robots available for 20 students, access would be limited so good design was imperative, the unit lasted 2 weeks out of a 14 week semester. The robots were built for the students and they had to navigate them around a 4-4 foot maze, the maze was not available until the final test, but unlimited access was available to the RCX programming environment. Students worked in groups of 4 to navigate the robot around the maze. At the end of
the two weeks the robots were available to the students for a period of 2 hours, after which the
group could modify their program but not retest. The solutions were demonstrated the following
week and were videotaped and even though no group achieved the goal valuable lessons were
learned. A reflective essay was required to be completed by each of the students, 19 in total, at the
end of the demonstration, in which the students answered a number of questions about the
experiment and their learning process. Outcomes based on the instructor’s notes, videotaped
demonstrations and the summative essays, indicate that the students did develop “an appreciation
for what it means to think through a problem before implementation”, and Wolz deemed the
experiment to be a pedagogical success and she plans to further integrate the robots into CS2,
possibly with a Java interface and other courses. This is an interesting use of robots where the
focus is narrow and the resources finite and indicates that not only can robots be used as a
concretising tool to demonstrate programming constructs but they can also be used effectively in
the design stage to reinforce good design principles and project management.
**Handy Car**

Stephen Linder and his colleagues at the State University of New York believe that “a context-based collaborative approach, combined with the excitement, motivation and real-world experiences provided by a robot, provide a nearly optimal method of teaching students how to acquire knowledge about computer science” (Linder et al, 2001). To this end they developed an inexpensive wheeled robot, the Handy Car (HC), built on a radio controlled car base, requiring only a simple set of tools for assembly and a Java-based server that allows students to issue commands to the robots and read sensor values from the robot through an abstracted Java interface. In addition the server has an Internet interface that allows clients on other PCs to logon and control the robot, facilitating the sharing of a robot between multiple students. The Handy Car can be used at introductory and advanced levels, enabling the execution of logical and conditional expressions, data structures for introductory levels and lending itself to the development of complex projects such as collecting balls, parallel parking and beacon searching and homing for the advanced courses.

Linder and his colleagues concluded from the use of the Handy Car that the students were invigorated and that it facilitated active learning, enabling the students to become more autonomous learners.

At Colby College a new course was offered in 2001, Exploration with Robots, which provided an introduction to programming in C and problem-solving skills (Goldweber et al, op. cit.). Students built simple robots and program solutions for problems such as keeping a robot inside an area marked on the floor, following a line, navigating around an obstacle. Students were able to design their own final projects, which included robots playing tag with each other, robots collecting objects, and robots that searched for objects, communicated the location of those objects to another robot, which would then navigate directly to the object. Congdon at Colby College concluded that the students were enthusiastic and creative in dealing with the robots.

**Benefits of Robots**

- Promotes active learning, engaging the student and promoting enthusiasm and learning
- Promotes collaboration and the robot becomes a participant in this collaboration (Linder et al, op. cit.)
• Provide experience with real machines
• Promotes creativity (Goldweber et al, op. cit.)
• Students can generate hypotheses and test these hypotheses, getting feedback immediately (Linder et al, op. cit., Barnes, op. cit.)
• Promotes good design and planning (Barnes, op. cit.)
• Promotes leading with practice, which fosters autonomous learning (Linder et al)
Robots – Pedagogy

Constructivist learning assumes that students acquire knowledge by constructing individual models of knowledge. Linder et al believe that not only do robots facilitate constructivist learning but they also provide experiences with real machines, not just simulated ones and allows the students to work in a complex environment making it more interesting for the students. Quoting Magnesen’s study of material retention rates based on modality of interaction, see Figure 1, and the fact that students need to retain more and more information due to the software libraries with which they must become familiar, Linder argues that students need a mechanism for retaining large amounts of material. The high retention rate of material obtained by “Doing and Discussing” and the fact that collaboration with other students forces students to organise their abstract ideas into concrete sentences promotes a deeper understanding of the concept under discussion. Linder and his colleagues argue that the use of robots facilitates this process as the robot becomes in part a participant, and that students feel empowered as they can produce complex behaviours from another physical entity. Students can generate and test their hypotheses, in this way retaining more information. These are principles that are inherent to the constructivism.

Given the increase in the use and interest of robots in the area of teaching programming and the relative youth of the robots it seems likely that more quantitative research will be conducted in this area. Patterson-McNeill & Binkerd(2001) is a useful resource for educators considering the use of Lego Mindstorms.
Problem Based Learning

Motivation

Boud and Feletti describe problem-based learning as

“a way of constructing and teaching courses using problems as the stimulus and focus for student activity. It is not simply the addition of problem-solving activities to otherwise discipline centred curricula, but a way of conceiving of the curriculum which is centred around key problems in professional practice”. (Boud & Feletti 1991)

According to Ellis et al (1998) problem-based courses begin from problems and that learning in these courses are driven by the problems, rather than the presentation of the subject content. The problems may come from the teacher or the student and these problems may vary from well-structured, for novice students, to open-ended and ill-structured reflecting the type of problems that the students will encounter as computing professionals. The problems themselves are “real-life” and cross traditional subject boundaries. The students work in small groups to solve these problems aided by a facilitator. Information on how to develop a solution is not usually given, but resources are made available that assist in the process of solving the problem which would include resources to facilitate group work, reference materials, resources to assist scaffolding such as visualisation and experimenting systems and problem solving software. The students have to determine for themselves what they need to learn in the relevant areas to solve the problem and they make use of the available resources to this end.

Ellis et al (op. cit.) argue that the computing discipline lends itself to problem-based learning due to the following factors:

- Computing is generally problem-driven
- Life-long learning is necessary due to the rapidly and continually changing nature of the industry
- The project group is the predominant mode of operation in the industry and
- Computing crosses discipline boundaries

The problem-based learning approach is a life-cycle process with the following steps, (Nulden, U., 1998)

- Challenge students with a problem
- Facilitate the group’s work on the problem
• Provide subject matter guidance
• Provide scaffolding to assist the students to gain new understandings within the framework of the problem and thus develop their independent learning skills
• Publish the student’s work on the problem
• Assess the students’ outcomes to provide feedback relating to the learning process through work on the problem.

“The intention of a problem-based learning approach is to allow students to develop skills for dealing with real-life problems throughout their careers and the ability to continue to learn effectively throughout their lives.” (Ellis et al 1998)

Project Development in an Introductory Course in Java
A programming course in Vannes University, concentrating on an object oriented paradigm and learning software engineering methodological concepts introduced a PBL sequence in the middle of the curriculum. All the students had to undertake a project and they were to work in pairs over a 4 week period. The language used was Java. The teaching method used in the rest of the course was a combination of lectures and labs, and Intranet technology to encourage self-training. The aim of the PBL sequence was for the student to develop a complete project from requirements identification, design, programming and testing, in the process increasing their skills in working methods, document writing, teamwork and project management.

The students were presented with the problem, being told that the software was actually needed by the departmental secretary in order to challenge them. The system being designed was a database system to register student or company information for mailing purposes. It included functionality such as input/output on a terminal; sequential and direct access files and menu-driven applications. The problem is then discussed in order to define the requirements, within a classroom setting with groups of 24 students.
The teacher presents the requirements document, which is not complete. The students’ first assignment is to complete the requirements document, in the process understanding the requirements and developing their writing skills. The students are then presented with the design
document, which is discussed in class and “role play” is used to help students visualize the “objects at work”. This ensures that the students have a complete understanding of the classes and their functionality and relationship with other objects and it is overseen by the teacher to ensure the process works correctly. In order to implement the required Java classes, new knowledge is required. The teacher discusses these classes and provides some specific information. The students then implement, document, test and validate each class.

Resources had been developed, from guidelines for the project to a Java class library dedicated to the programming exercise and made available to the students. A specific newsgroup was created for the project to facilitate communication from teacher to student and vice versa. As the project progressed students offered solutions to other students’ queries indicating increasing motivational levels. The students delivered their finished products electronically. The work of the student pairs was evaluated by

- Evaluating the produced documents
- An oral presentation of project, including demonstration of the generated classes
- Individual interviews.

**A PBL Trial in First Year Computer Science (Greening, Kay & Kingston 1997)**

In 1996 the Basser Department of Computer Science at the University of Sydney ran a PBL trial parallel with its first year programme. There had been “long standing disgruntlement” with the use of Pascal as a first year language in the department and this was changing in 1996 to an object oriented choice, their own in-house developed OO language, BLUE. A proposal was put forward to use a PBL approach to teaching which was met with some concern but it was agreed that a trial would take place in which a few classes of students would do a PBL version of the 1996 course (Pascal) in parallel with the conventional approach. This would then be evaluated in reconsidering the proposal.

Initially the students were introduced to the concept of PBL, basic UNIX skills, initial programming in Pascal and group working. During this period the students were given two problems, create a homepage and write a minimalist UNIX manual. The tasks, which were essentially individual ones, were tackled in a group environment and overcame the initial group fluctuations at start of the semester.
Then the students were given a large group-based problem to solve in which students were required to develop and implement plans for researching, designing and coding solutions to a chosen problem, cryptology scenario or videoshop database. Progress reports were required at different times and code reviews were carried out by the tutor and all the members of the group. Each individual was required to take responsibility for some specific coding aspect. Finally the students reflected on their learning and presented their work to the rest of the class and for assessment. The presentation consisted of reports documenting their work, including individual reflective statements, code submissions and a demonstration of their product to the class and staff. The semester work was supported by a weekly three-hour workshop and a further two-hour period was used for activities designed to facilitate the development of certain skills important to the success in the problem-based activities, group-working and technical skills. An evaluation was carried out on the trial and yielded the following results:

- In the overall written exam scores no significant difference was revealed between the PBL and conventional group
- In the practical exam the performance between the two groups were not statistically different
- Surveys indicated that only 5% of the PBL students felt the course overwhelming compared to 30% of the conventional group and the PBL group felt more positive about the course than the conventional group.
- The trial motivated and facilitated the PBL group in learning other skills, report writing, verbal skills, planning and research skills.

Although the trial was not without its problems, e.g. the developers saw the need for more self-assessment tasks, and an improvement of the administration of the course, it did lead to the department adopting a PBL approach in first year computer science, together with an object-oriented language.

**A Problem-based Interface Design and Programming Course** (Kay & Kummerfeld, 1998)  
A course based on problem-based learning that interweaved interface design and learning programming was run by the authors. The students worked on “authentic” problems in groups and the tasks given required inquiry, information gathering and reflection. The course was more structured than usual in a PBL scenario, as students were required to tackle the problem in several stages and each stage involved 2 lectures on a new programming tool, 2 lectures on a new aspect
of interface design, 2 practical classes and private study. Students worked in groups but were expected to develop individual solutions for discussion. Each stage involve web programming prototype, screen design and python prototype, usability and X-toolkit prototype, user-centered design and Java or VB prototype, reflective report explaining and assessing their design. Each prototype consolidated learning, as it involved a new programming tool and a new element of design. The students learned to create their prototype by studying an example program. There was significant group-based assessment in the course. Evaluation of the course showed that some groups did considerable research into similar classes of systems as their own, analysing user trials and were aware of the shortcoming of their designs. Where the course is structured around a single problem which drives the learning, and the exposure to many programming tools, which is a reflection of what many of the students will face when in the working environment, there is no empirical evidence to suggest this approach is more effective than a traditional approach to teaching. Without such evidence and just a student questionnaire to go on, where the majority of the students were positive about the course, we can draw no clear conclusions about the educational benefits of the approach taken.

**A PBL program for an advanced software engineering course.**

McCracken and Waters (1999) felt that software engineering suffered from the “*dual problem of exponential increases in the knowledge required to practice it and in the complexity of the problems it is expected to solve*”. They also acknowledge that the educational infrastructure is constrained by a constant level of instructional hours with which to address these problems. Given the success of the PBL approach in medicine they decided to implement a PBL program for an advanced software engineering course.

Their implementation followed the following methodology

- All students were in a PBL group
- The problem was a large “unconstrained” problem, it was ill-structured both to students and facilitators
- Problem-related materials were difficult to obtain. Students interacted with real customers
- Single problems illustrated multiple areas
- Part time facilitator available at one meeting per week
However their results indicated a bimodal distribution of students, one peak that felt that PBL was the most effective teaching method that they ever participated in and the other which felt that PBL was the most useless experience in education that they had ever had. McCracken & Waters also observed the following

- Students didn’t develop deep learning issues
- Students focused on problem solution concerns and not on the process
- Students didn’t develop the desired metacognitive skills
- Students could not set effective learning goals, focusing on specific computer-related rather than deeper issues for their individual learning.

To understand why the PBL process didn’t produce the same success as that in the medical world they conducted an ethnographic study of the students and they observed the problem of collaborative learning, where the students had problems in team working and tended to avoid problems rather than resolve them, indicating the need to provide students with group dynamics skills. They noted the need to aid students in identifying deeper issues for their individual learning.

They conclude that more direct scaffolding of the goal-setting process is need early on in the course to develop these skills. They also suggest providing additional scaffolding via group discussions on product versus process issues to help students learn to identify the difference. The lack of support from an expert facilitator was recognised as an issue, which detracted from the PBL process.
Benefits of Problem-Based Learning

- Fosters life long learning techniques by forcing the student to reflect on their learning process and re-evaluate it, through maintaining learning diaries and portfolios
- Promotes understanding through collaborative work
- Learning is driven by the problems rather than subject content and reflects real-life problems, preparing students for the working environment (Ellis et al, op. cit.)
- Problems can vary from structured to ill-structured to cater for novice to advanced students
- Promotes creativity in deriving solutions to problems
- Promotes team-working skills (questioned by McCracken & Waters, op. cit.)
- Promotes independent learning and forces students to take responsibility for their work
- Promotes positive feeling about course (Greening, Kay & Kingston, op. cit.)
- Students learn other skills not specific to course, verbal, report writing, demonstration (Greening, Kay & Kingston, op. cit.)

Problem Based Learning - Pedagogy

“PBL adopts a constructivist view of learning, in which learners make continual adjustments to existing belief constructs. This approach favours the provision of learning experiences which are owned and defined largely by students who negotiate the meaning of such experiences in small peer-based groups” (Greening et al, op. cit.). Greening et al found that it “provides a natural basis for addressing “missing” generic skills (ability to work in small groups, act as independent learners which are required of all graduates of the University but tend to escape course inclusion”. The active engagement of students in the construction of their knowledge is one if the core tenets of constructivism. Ellis (98) argues that the use of electronic tools, which allow feedback, was prompted by the early Skinnerian teaching machines. Another tenet of constructivism is collaboration which is central to problem based learning, which allows the group construction of knowledge and is now facilitated by electronic communication and collaboration tools for different patterns of communication, one-alone to many-to-many. Constructivist learning focuses on initiative thinking activities and students generate their own strategies for problem formulation and solution. Therefore problem based learning promotes metacognition through encouraging students to reflect upon the problem-solving process, (Vat, 2001). The production of
learning diaries and portfolios aid the students in developing these reflective and self-analytical skills, (Ellis et al, op. cit.).
Cognitive Apprenticeship

Cognitive Apprenticeship is a model of teaching based on a fact observed by researchers; “in the learning of expert behaviour, too little attention has been paid to the processes applied by the experts in their activities. Cognitive apprenticeship training aims at teaching these processes” (Enkenberg 2001). This comes from the traditional apprenticeship model which emphasises observation, training and practice, initially with a master and later independently. The method is characterised by the fact that application takes place in a social setting. The problems are selected on pedagogical grounds and teaching emphasises contextualisation of the things to be learned so that the resulting knowledge may be applied in different content fields. (Enkenberg op. cit.).

Webworlds

The advent of the WWW and Java has led to a proliferation of interactive applets and software libraries for educational use that encourage an experimental and modelling approach to learning (Chalk 2000). Chalk states that by combining Papert’s description of a microworld as “an active learning environment” with the availability of these facilities on the Web the term “Webworlds“, i.e. web-based interactive learning environments, has been developed. He argues that the provision of Webworlds and the opportunities to use or develop software environments on the Web can “apprentice” students into a community of practice of software engineers, if collaborative and other tools properly scaffold the process.

Chalk describes three tools on the Web, JSP Editor (Engström, 1997), Java Flowchart Editor (Cumberbatch, 1999), and Visualizing Graphs with Java (Barowski & McCreary 1998), that encourage participation in the processes of software engineering. By providing these experimental and modelling tools and their support by experts the apprentice student may use them as part of the curriculum or as a learning resource.

Chalk conducted pilot studies of the use of Webworld resources, specifically JSP Editor, in introductory software engineering courses. Students used the JSP editor to express their models and then the JSP Editor would produce Pascal or C code. The students could exchange comments with their tutors during the course of the assignment. The changes made to the stored diagram or model were analysed in terms of the “learning paths” they followed. He conducted this research in two phases, one where the Webworld was made available as a resource and second where it was an integral part of a data structures and algorithms course. Chalk then conducted a qualitative case
study using observation, interviews, student diaries, assignment reports, on-line records and questionnaires and tests. Results from the first phase where the author provide support and the Webworld was used as a resource, “something that might assist them doing JSP”, indicated a positive response overall from the students but the post-test results did not produce very good results and student diaries were not filled in contemporaneously. Observation time was insufficient to provide data about learning patterns. Chalk decided to use a collaborative workspace on the Web to record this data.

The second phase was implemented using a shared forum, BSCW (Basic Support for Co-operative Work, Horz Informatic 1997), and the students worked in pairs, each pair given a private team area. BSCW could be used for the provision of feedback in the form of tutor comments as the project developed. Students were encouraged to copy their diagrams into the BSCW web space and communicate with their tutor and each other about their solution. Student interactions could be monitored and responded to using BSCW at least every day. Pre and post tests and questionnaires were administered, interview and observation notes kept and assignment reports copied. Chalk argues that the use of the appropriate tool and the assistance of tutor and peer feedback enabled the students making the best use of them to achieve the best results. Chalk concluded that the Web space monitoring assists the assessment of the process of assignment. Considerable use was made of the communication and collaboration facilities provided by BSCW, which Chalk argues is evidence in support of guided or scaffolded resource-based learning in the apprenticeship model. However results from an assignment using data structure Webworlds were not encouraging. The tools were not as easily accessible, the models were exploratory rather than expressed by the students themselves and the assignment objective (to compare and contrast a variety of data structures) was much vaguer than the previous assignment. As a result there was no evidence of creative activity until the day of the deadline.

Chalk concluded that “the Web, collaborative support and modelling tools together provide the means for a virtual apprenticeship into the practice of software engineering”.

**A Multimedia Learning Environment for Object Oriented Design**

A project at the University of North London aims to develop a set of pedagogically sound techniques for learning object-oriented design and to develop a multimedia-learning environment based on those techniques (Boyle & Yazici). The pedagogical approach for the environment is based on Constructivism, in particular guided discovery variants such as CORE (Context, Objects,
Refinement, Expression) and Cognitive Apprenticeship. Boyle argues that two significant challenges for learners of object-oriented design are abstraction and complexity and that his learning environment tries to overcome abstraction by using visual concrete realisations, and complexity is dealt with by providing a scaffolding mechanism, an “assistant” that provides expert help, tips or demonstration.

**Benefits of Cognitive Apprenticeship**

- Promotes metacognitive skills by promoting self-reflection and self-analysis
- Provides scaffolding – the expert can guide the student with hints, demonstrations, gradually withdrawing support when appropriate
- Promotes user-led learning
- Feedback is available to guide students
- Teachers can determine the process of the student’s thinking thus enabling the teachers to determine where problems lie for students and enable them to provide the necessary scaffolding for that problem area
- Promotes collaboration and learning in a social setting, with experts and other apprentices
- Availability of tools on the Web facilitates the development of apprenticeship learning
- Promotes development of other skills, verbal and communication

**Cognitive Apprenticeship - Pedagogy**

The underlying model of cognitive apprenticeship is the traditional apprenticeship model where the apprentice learns from the processes applied by the expert. It is a collaborative teaching model where the emphasis is on the support of the construction of knowledge (Ekenberg, op. cit.). According to Enkenberg cognitive apprenticeship applies several teaching and learning strategies:

- Modelling, i.e. the demonstration of the processing of thinking
- Explanation, why activities take place as they do
- Coaching, monitoring, assisting and supporting students’ activities
- Scaffolding, support of student in order to cope with the task in hand and the gradual withdrawal of the teacher from the process.
- Reflection, self-assessment and self-analysis
• Articulation, results of reflection put in verbal form
• Exploration, students are encouraged to form hypotheses, test them and find new ideas.

The basic tenets of cognitive apprenticeship tie in with those of constructivism, the student constructing his own knowledge, aided by an expert initially and gradually becoming an independent learner, the collaboration aspect and the development of metacognitive skills in reflecting on his work. The social aspect of learning, as expounded by Vygotsky, is inherent in the cognitive apprenticeship model.
Miscellaneous

There is work ongoing in a number of other areas to determine a better way of teaching a programming course.

Approached based on Constructivism

Using a student learning model expounded by Laurillard, Davy & Jenkins (1999) integrated a discursive component into their course by means of small-group weekly tutorials, which focused on discussion-based exercises. They incorporated an adaptive component into the course by tailoring exercises through feedback of student progress. An explicit reflection component was introduced, as students were encouraged to reflect on their motivation for learning to program, and reflect on the components of the course. Acknowledging research that links heavy workloads and “surface” learning, the amount of assessed work was reduced by about 50%. All these initiatives, when evaluated, indicated that student learning had improved.

At the University of Sydney a range of innovative techniques have been introduced that encourage students to reflect on the state of their knowledge and on the process by which they acquire it (Fekete et al 2000). Fekete and his colleagues acknowledge the importance of reflection in assisting students develop metacognitive skills. For example, by explicitly outlining subject goals, getting the students to maintain a reflective diary and awarding points based on reflective activities in assignments and/or exams, students are encouraged to think about what they know, how they learned it and how well it matches the announced goals of the subject.

Looking to research that indicates that cooperative or collaborative learning models enhances academic achievement, the University of California carried out a study to investigate the effects of pair-programming on student performance in an introductory programming class (McDowell et al 2002). They concluded that students who programmed in pairs produced better programs, completed the course at higher rates and performed as about as well on the final exam as students who performed independently. They suggest that pair-programming can be used effectively in an introductory programming class (McDowell et al, op. cit.). An interesting outcome of the study was the higher finishing numbers of students in the pairing group (92% vs. 76%) indicating that pairing may increase the likelihood that students complete the introductory programming course.
Backed by research that suggest that programming expertise is partly represented by plans, templates, schemas, Clancy and Linn (1999) at the University of California surveyed research results relating to the understanding and use of patterns in introductory programming courses. They concluded that design patterns should be created so that students can connect them to new problems; the patterns also need rich connections to examples and multiple links to the context of use, by the use of case studies and by class discussion involving other students and the instructors. Enkenberg (op. cit.) states that case-based teaching seeks to develop analytic reasoning, collaborative action and communication skills. Using patterns with certain types of exercises, such as comparison of related patterns, reusing code, code comprehension, Clancy & Linn (op. cit.) conclude it will benefit the students and believe that the social context of programming classes will reinforce pattern use and understanding. The use of patterns support students in integrating their knowledge of programming.

Gray et al(1998) used a constructivist approach, as exemplified in the CORE methodology, as a basis for a Java learning environment. The components of the CORE approach are

- Context - introduces a context for the learning material
- Objects - offer examples of the topic and demonstrate what the students should have acquired on completion of the learning activity. Students are able to interact with such objects.
- Refinement - students are offered further examples of the topic and through interaction, questions and feedback the student extends and refines their understanding of the topic.
- Expression - the student should be able to express their understanding of the topic and this is reinforced through explicitly stating what has been learned and through practice of the skills.

The web-based system was developed in Java to teach people how to program in Java, offering students a variety of ways to interact with the material, sequentially or randomly, to cater for novice or experienced students. Navigation through the material was flexible, audio and video material was included, and “hot links” to other topics. Questions were included to enhance the students understanding and once a student had answered certain questions correctly they are allowed proceed to further topics. The user can control how they navigate through the material, and they can choose whether they need further examples(Objects) or wish to proceed to the questions and feedback section (Refinement). Video clips show an expert explaining the operation
of the examples. Users can see an animated step through of example programs where there is combined use of video and program output to show the effect of individual statements. The system was evaluated with regard to its effectiveness as a learning environment. The students were given two chapters of the CLEM(CORE Learning Environment of Modula-2) system and asked to compare the two systems, the content being identical, selection and simple iteration. Student feedback was gained through observation of the interaction with the systems, questionnaires, and a debriefing of students after they had worked on the systems. The students were a combination of undergraduates and postgraduates. The results were “extremely encouraging”, students appreciated the greater level of control over the order in which they could cover material. While the questions were seen as very helpful in checking how far the students had progressed with the learning aims of a section, the students criticised the level of feedback in the system, looking for reasons “why” an answer was incorrect and an inclusion of links back to relevant learning material so that the student could check their understanding.

**FLINT**

Looking at the environments which novice programmers use when learning to program lead Ziegler and Crews (1999) to develop their own integrated development tool as they felt the commercially available IDEs (integrated development environments) did not best support novice students trying to learn programming. To this end they developed FLINT, an instructional program development environment for use in the first semester of a CS1 course, which supports design, a top-down approach that is recorded in a structure chart, algorithm development, using structured flowcharts and testing and debugging with the students getting immediate feedback on their work. Their design is saved and the lecturer can gain insights into the student’s design habits and work process and may also be used for reflection by the student after the project is complete. No high level language is introduced until students have designed and implemented and evaluated programs of significant size, thereby giving the students a “concepts first” approach to software development.

This project supports the constructivist approach to knowledge acquisition by allowing the student construct their knowledge in an active way, providing feedback for self-analysis and reflection.
Educational Theory

While researchers grapple with the problem of teaching programming and the development of new tools one must bear in mind that what we trying to do is teach and aid the student in his learning, we therefore must acknowledge the need for a sound pedagogy underlying any teaching approach we may adopt. Powers & Powers (1999) argue that “no one method is a panacea, rather the best teaching occurs through a synthesis which incorporates a multiplicity of methods to enfranchise the greatest number of learners”. They argue that the methods used should be tailored for the instructional context, including the topic, demographics of student population and resources etc. The theory of constructivism states that all our knowledge has been constructed from our own personal experiences and social interactions in a particular cultural setting. Any new knowledge that the students construct, in response to a new experience, will be incorporated into the framework of knowledge that they already have. An implication of this theory, according to Powers & Powers (op. cit.), is that our teaching must be experiential to be effective. This could be applied to AV technology where the learners actively engage with the visualization. Students must also be provided with a social setting in which to learn, an opportunity to form new knowledge in cooperation with their peers. The notion of experiential, hands-on learning in computing is well recognised and implemented in most curricula. The view that knowledge is constructed in a social setting has been widely adopted in most faculties where collaboration and team work is widely adopted, giving students valuable experience for when they start their employment as project team members.

Discovery Learning describes any activity in which the learners are free to make their own discoveries about a certain phenomenon. There has been a more narrow definition where the students initially are allowed to spend time doing “unguided exploratory work”, followed by a stage where the students are given formalized options to investigate followed by a final stage where the class is brought back together to summarize their findings and use them to answer questions. This approach in Computer Science seems to be rare and more work is associated with a problem-based learning approach, where the students are the active problem solvers and the lecturers “assumes the role of cognitive and metacognitive coaches”. While dealing with ill-structured, real-world problems the students become independent, flexible and self-directed learners. A number of computer educators, (Kay, McCracken), have adopted this approach in
differing forms and Powers & Powers (op. cit.) see potential for adoption of problem-based learning in the proliferation of design problems throughout the curriculum.

While many computing educators will claim to be using cooperative learning, researchers claim that there is more to cooperative learning than using groups to teach team work. Other features that are necessary are

- individual learning objectives
- individual accountability
- positive independence- no one student can carry the group
- heterogeneous groups
- instructor provided rules of social engagement
- post group evaluation.

Another factor that we must bear in mind in teaching is learning styles - strengths and preferences in the ways that students take in and process information, which could be factual or theoretical, visual or verbal or actively or introspectively. Felder argues that functioning effectively in any professional capacity requires working well in all learning style modes (Felder 1996). He argues that an objective of education should be to help students build their skills in both their preferred and less preferred modes of learning.

The Myers-Briggs Type Indicator (MBTI) model classifies students according to their preferences on scales derived from Carl Jung’s theory of psychological types. Students may be

- extraverts(try things out) or introverts(think things through)
- sensors(practical, detail-oriented) or intuitors(imaginative, concept-oriented)
- thinkers(sceptical, base decisions on rules, logic) or feelers(appreciative, decisions based on personal or humanistic considerations)
- judgers (set and follow agenda) or perceivers (adapt to changing circumstances).

A student may be an ESTJ(extravert, sensor, thinker, perceiver) while another may be an INFJ(introvert, intuiotor, feeler, judger), the type preferences can be combined to form 16 different learning style types. A traditional lecture with individual assignment requirements would be oriented towards introvert while if we develop a more active teaching style with cooperative learning we would cater for the extroverts.

The Kolb learning style model classifies students as having a preference for (a) concrete experience or abstract conceptualisation and (b) active experimentation or reflective observation.
There are four types of learners in this scheme:

- concrete/reflective – the student who asks “why are we doing this?”. To be effective with this type of student the lecturer should act as motivator.
- abstract/reflective – the student who asks “What” are we doing? – the lecturer needs to act as an expert to be effective with this type of student.
- abstract/expert – “How can we apply this?” – the lecturer needs to act as coach with this type providing guided practice and feedback.
- concrete/active – the student who asks “What if” – the lecturer should allow the student room to experiment with new situations and room for self-learning with this type.

To reach all types of learners, the lecturer should explain the relevance of each new topic, present the information related to the topic, provide opportunities for practice in the topic, and encourage exploration.

The Felder-Silverman Learning Style Model classifies learners as:

- sensing (concrete, practical) or intuitive (conceptual, innovative, oriented towards theory)
- visual (prefer visual representations) or verbal (prefer written or spoken)
- inductive (specific to general) or deductive (general to specific)
- active (trying things out, working with others) or reflective (thinking through things, working alone)
- sequential (linear, orderly, learn in small incremental steps) or global (holistic, learn in large leaps).

According to Felder teachers and students may prefer one side of a dimension in one subject and the opposite side in another, but generally they prefer one side or the other in most subjects. Researchers believe that if an instructor “Teaches around the Circle”, i.e. the learning needs of students in each category are met at least part of the time, the needs of all the learners are met. Felder argues that if instruction is balanced on each dimension of any of the above learning models, then this meets the needs of essentially all the students in the class. When adopting new methods to teach programming, we should bear in mind the learning style to which it is oriented and try to balance its use with other methods that appeal to the other dimensions in the learning style model, e.g. if we focus on visualizations we need to supplement these with verbal and written explanations in lectures and readings.
Gardner’s (1993) Multiple Intelligences theory argues that there are several (now 9) types of intelligence: linguistic, logical/mathematical, musical, spatial, bodily-kinaesthetic, interpersonal, intrapersonal, naturalistic and existential. Each person is endowed with varying levels of intelligences across the different areas. In computing one would expect to see students with high logical intelligence but students could apply their other intelligences in different areas of the course, spatial in designing program flowcharts, or GUI design.

Sequencing plays an important role in the teaching of programming and it is sometimes determined by the textbook of the course. As lecturers we should have adopted a rationale for the sequencing that we follow. It may be based on Bloom’s taxonomy of learning objectives, moving from simple levels of knowledge, comprehension and application to higher order processes of analysis, synthesis and evaluation. Lister(2000) states that this taxonomy has had only limited recognition in computer science education and he argues against having students writing complete programs as early as possible as this skips to the fifth and sixth level when these levels are based on the preceding four. We could alternatively order the curriculum from the more concrete topics to the abstract, according to the work of Piaget. Sequencing suggests that programming with concrete objects such as robots or physical props could be useful before moving on to abstract environments (Powers & Powers (op. cit.)), although Papert argues for a “re-evaluation of concrete”, suggesting that abstract reasoning is not more valuable than concrete manipulations.

Whatever approach to teaching programming we adopt we must not lose sight of our primary goal, to teach and therefore we must base our course on a sound pedagogy, bearing in mind the audience we are teaching to and the variety of learning styles, abilities and backgrounds from which they come.
Discussion

There is no doubt from the available research that learning to program poses a problem for the undergraduate student. As an acknowledgement of this problem research has been active to seek a way to improve the teaching and learning of programming. As a result a number of approaches have developed, from the technology driven which involves the use of software visualisations, design tools and robots to the educationally driven paradigms such as problem based learning, cognitive apprenticeship and case or pattern based learning. At present the majority of approaches are seen as complementary to the traditional lecture and lab scenario, being integrated in some form into a structured guided teaching approach.

The approach that has seen the most research to date is software visualization. A number of issues have risen out of the research in these areas, for example the initial mixed results from studies regarding the effect of visualisations on student learning (Wiggins, op. cit.) and the need for more studies with larger groups to be performed. Wilson & Aiken (1996, op. cit.) surveyed how several algorithm animation systems are used in computer science education in order to determine how these systems can be most effectively used for teaching purposes. While they noted the increase in usage of visual animation systems in classroom instruction they found that it was difficult to obtain detailed information about how the systems are being used. Formal studies carried out Wilson & Aiken (op. cit.) implied that for animation systems to be used most effectively the students must first, be actively engaged with the tools, e.g. by designing their own datasets, and second, the animations should be included as “part of an instructional context which also includes textual components and which features carefully focused task assignments designed to achieve specific learning objectives”. Wilson & Aiken point out that guidelines for the design and use of animations are needed so that these animations become effective teaching tools and not just entertainment for the students.

An overview of visualisation, its use and design, was published by the Working Group on Visualisation (Naps et al op. cit.) in which it presented a discussion on when the use of visualisation is most appropriate and design issues for instructional visualization tools. The report states that some form of visualization, either picture or animation, should be used in teaching computer science concepts and algorithms, the choice of visualization depending on topic and cost-effectiveness. Pictures are appropriate when the amount of data is small and the data structure is very simple and an animation can be used when the quantity of data is large, the data structure is
complex or when movement is needed to show how the relationships between objects change over time. The decision to generate a hand drawn or computer generated visualization depends on the cost benefit analysis. Naps et al argue that visualizations can be used in classroom demonstrations, laboratories and traditional assignments with the student being a passive viewer or active participant in instructor-led, individual or collaborative work. He cites evidence that active engagement of the student leads to higher motivation and better retention of content. However the educational benefit of using visualizations has been mixed, particularly when the student is not actively engaged, with only 13 out of 24 experimental evaluations of AV effectiveness showing that some aspect of visualization technology significantly impacted learning outcomes (Hundhausen(2002)).

In experiments carried out in 1996 Stasko concluded that the “benefits of animations are not obvious” and that researchers should carefully determine the “specific pieces knowledge that an animation can help a learner acquire or practice”. Stasko (1999) carried out an observational study in order to determine how learners utilize animation in trying to understand a new algorithm and how animations can fit into successful learning strategies. The study focussed on cognitive aspects of the domain, as the authors felt that while the technology for AV was advancing at a fast pace, understanding of the educational benefit of using the technology was not matching this. He found that algorithm animations made a challenging algorithm more accessible and less intimidating which lead to enhanced student interaction with the materials and facilitated learning.

The lack of uptake of usage of these tools, many of which are freely available on the Web, is a concern for researchers and raises the issue of ease of use of the tools and the learning curve for the student and lecturer in incorporating these tools into the curriculum. The creation of visualization tools is also an issue as it is time-consuming and difficult (Naps et al, 1996). The Working group on Visualization call for an effective means of sharing these tools, such as a web based repository which would provide

- a single location for tool users and developers to investigate the existing set
- provide tried and tested visualization tools for anyone to use
- facilitate new development to build upon previous work
- encourage new development
A general consensus seems to be that visualisations need to be interactive, support reflection and supported by as much textual documentation as possible. Researchers acknowledge that more work needs to be done to generate and use stop and think questions thereby encouraging the students to analyse what is happening in the animation. Jarc et al (2000) questioned the educational benefit of the use of interactive predictions in AV but Roßling and Naps (2002) cite a study where the predictions are part of a “quiz-for-real” the students are “forced to take” and it shows that they do have a beneficial impact on the learner, rather than answering stop-and-think questions which were merely used for immediate feedback. Also more work is needed on the dynamic production of context-sensitive documentation to take into account the data to which it relates (Naps et al, op. cit.).

Hundhausen et al (2002) conducted a meta-study of 24 experimental studies of AV educational effectiveness, which the authors claim is close if not the entire population of published AV effectiveness experiments. The aim was to attempt to address whether AV technology is educationally effective A synopsis of these studies is provided in Table 5. Eleven studies showed a statistically significant difference between the performance of a group using some form of AV technology over another group using either an alternative form of AV technology or no AV technology at all. 10 studies showed no significant result and two studies found a significant result in which the positive impact of visualization could not be disentangled from another factor. One study yielded a negative result where the students who used the AV technology actually performed significantly worse than the students who used text-based tools. The results substantiated the pedagogical impact of using AV technology to engage students actively in their own learning leading the authors to conclude that their most significant finding was that “how” students use AV technology has a greater impact on effectiveness than “what” AV technology shows them. The meta-study found that in studies where the students merely viewed visualizations no significant learning advantage was gained over students using conventional materials. The most successful educational uses of AV technology, as suggested from the meta study, are those where the technology is used for actively engaging the student in the process of learning, by means of what-if analysis, prediction exercises and programming exercises, leading the authors to suggest that this form of AV technology enables the students to construct their own understanding of algorithms through a process of active learning.
The Working Group on “Improving the Educational Impact of Algorithm Visualization” (Naps et al, 2002) argue that such technology is of “little educational value unless it engages learners in an active learning activity”. The Working Group have developed a taxonomy of learner engagement with this technology and based on Bloom’s taxonomy they suggest metrics for assessing the learning outcomes to which this engagement may lead. These taxonomies of engagement and effectiveness metrics provide a framework for experimental studies of visualization effectiveness which they hope will be taken up by educators.

The Working Group and its members carried out three surveys of computer science educators to determine current practice in the field. Certain questions attempted to determine the participants attitude to visualization, the usage of dynamic visualizations to support learning and typical class set-ups. The results indicate that while there is a strong perception among educators that visualizations can help students learn, this is not borne out in the research and few use dynamic visualizations frequently, only 3% use visualizations every week from one survey, and even among those who use visualizations, few tightly integrate them into their courses. The way the learners use the visualizations ranged from, not directly using the visualization, to construction of visualizations as homework, as tools available for construction of own visualizations while none were asked to present their own constructions. The most common usage of visualization, from one survey, indicate that students watch the visualization in the class (90%), followed by usage of the visualization in the labs (72%). These usages were mainly complemented by oral class questions (52%) and written open-ended type essays (45%). The most common type of course where dynamic visualizations are used was data structures and introductory programming was mentioned by more than one respondent. The respondents indicated the benefits they had experienced from using visualizations, as shown in Table 6:
<table>
<thead>
<tr>
<th>Percentage</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>The teaching experience is more enjoyable</td>
</tr>
<tr>
<td>86</td>
<td>Improved level of student participation</td>
</tr>
<tr>
<td>83</td>
<td>Anecdotal evidence that class was more fun for students</td>
</tr>
<tr>
<td>76</td>
<td>Anecdotal evidence of improved student motivation</td>
</tr>
<tr>
<td>76</td>
<td>Powerful basis for discussing conceptual foundations of algorithms</td>
</tr>
<tr>
<td>76</td>
<td>Allows meaningful use of available technology</td>
</tr>
<tr>
<td>72</td>
<td>Anecdotal evidence of improved student learning</td>
</tr>
<tr>
<td>62</td>
<td>(mis)understandings become apparent when using visualizations</td>
</tr>
<tr>
<td>52</td>
<td>Objective evidence of improved student learning</td>
</tr>
<tr>
<td>48</td>
<td>Interaction with colleagues as a benefit</td>
</tr>
</tbody>
</table>

Table 6: Benefits of using Visualizations (Source: Naps et al 2002)

Participants were asked about factors that make them or their colleagues reluctant or unable to use dynamic visualizations, the results from one survey, also reflected in the other survey were as follows:

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>Time required to search for good examples</td>
</tr>
<tr>
<td>90</td>
<td>Time it takes to learn new tools</td>
</tr>
<tr>
<td>90</td>
<td>Time it takes to develop visualizations</td>
</tr>
<tr>
<td>83</td>
<td>Lack of effective development tools</td>
</tr>
<tr>
<td>79</td>
<td>Time it takes to adapt visualization to teaching approach and/or course content</td>
</tr>
<tr>
<td>69</td>
<td>Lack of effective and reliable software</td>
</tr>
<tr>
<td>66</td>
<td>Time it takes to transition them into the classroom</td>
</tr>
<tr>
<td>66</td>
<td>Unsure how to integrate the technology successfully into a course</td>
</tr>
<tr>
<td>59</td>
<td>Lack of evidence of effectiveness</td>
</tr>
<tr>
<td>55</td>
<td>Concerns about the equipment or presentation location (e.g. darkened room)</td>
</tr>
<tr>
<td>48</td>
<td>Unsure of how algorithm visualization technology will benefit students</td>
</tr>
<tr>
<td>38</td>
<td>Students are too passive</td>
</tr>
<tr>
<td>31</td>
<td>AV may hide important details and concepts</td>
</tr>
</tbody>
</table>

Table 7: Factors that impede the use of visualizations (Source: Naps et al 2002)
The results of the surveys show that computer science educators are convinced that visualizations can make a difference in "helping learners better learn concepts". The most frequent use of visualizations is for demonstrations purposes during lectures, the next usage being making the visualization available for student use outside of a closed lab setting. Only about half of the survey respondents ask their students to answer written or oral questions about the visualization leading the Working Group to conclude that visualizations are not tightly knit into the structure of the course. The authors felt that the key result of their surveys was that the major impediment to using visualizations was the time it takes to develop them. The surveys also showed that the evaluation of visualizations effectiveness was largely based on informal feedback or brief questionnaires, therefore primarily anecdotal.

Naps et al(op. cit.) cites these results in the development of an engagement taxonomy, (No Viewing, Viewing, Responding, Changing, Constructing to Presenting), recognising that active engagement with the visualizations is more important than the graphics that they see. Naps et al (op. cit.) also define a basis for defining metrics for determining the effectiveness of visualizations using Bloom’s taxonomy. By defining this framework for future research into the effectiveness of AV technology the Working Group acknowledge that active learner engagement is the key to gaining educational benefit from the technology, which concurs with the Constructivists theory.

The Working Group’s findings are interesting as they give us an appreciation of how AV technology is being used in academic settings, the most common usage being for demonstration purposes in lectures which is a passive mode from the learner’s point of view. If we take on board the findings of Hundhausen (op. cit.) where AV technology was seen to be educationally effective if it was used to actively engage the learner then we need to rethink how we are using the technology in the curricula. To be educationally beneficial it needs to be more tightly integrated into the curriculum with the learners being actively engaged with it. This implies that the learners should be using the AV technology for changing, construction and presentation purposes and they should be questioned about their usage and learning with the technology. Another implication of the Working Group’s findings is the concern that academics have over the time that is involved in using the technology, to overcome this concern the tools available need to be easy to use and install and not pose another learning curve on staff or students which would detract from the core concepts that the students are trying to learn. Other factors that we must bear in mind is that
students learning styles differ and several models have been presented in the context of computing education, e.g. the Felder-Silverman learning model and Kolb’s learning model. The Felder-Silverman model defines different dimensions of learning and each learner’s position in the dimensions can affect their learning when different learning methods and tools are used. The learning styles can blur the results of evaluation studies depending on the representation of types of learners in the population being evaluated (Naps et al 2002).

The results of the formal studies into the use of Creator, where a combination of two technologies, programming by demonstration (PBD) and visual before-after rules was used, concluded that students (children) who use it first and then move on to a traditional language are better programmers. This lead Smith et al to suggest that this approach is more acceptable to novices than traditional approaches. However this tool is limited and would need to be developed for undergraduates. Other visual environments, SIVIL, are being developed and evaluated currently. Anecdotal evidence would indicate that the use of Visual Basic is increasing, however little published work exists as to it’s educational effectiveness.

Using a 3-D interactive animation environment as a primary instructional tool for an introductory course raises a number of issues. These were acknowledged and categorised by Dann et al (2000a):

- **Graphics Concepts**
  Although the developers of 3-D environments provide built-in methods that support object positioning and motion, the students still needed an understanding of the coordinate system and the spatial relationship of objects to one another. This becomes more complex as objects move around and may or may not be aligned with other objects.

- **Program and Language Constructs**
  The flow of execution is complicated by the fact that some animations depicts simultaneously running processes.

- **Events and Responses**
  It was found that the greatest challenge was in teaching the students how the control panel was linked to the response (Dann et al (2000a)).

While the benefits of using robots have been acknowledged, their use does generate a number of issues. The students may find it difficult to link the control flow within the driving program and
the sequence of actions that the physical model goes through i.e. their concurrent and event-driven nature, which is basically the same issue as with 3-D visualisations. The fact that the amount of memory in the processors is small and thus the firmware and API need to be lean which means that the environment could distort good object oriented programming style (Barnes, op. cit.). The fact that with only the single LCD line on the processor to print to if things go wrong the lack of feedback can be a problem. The length of time taken by the edit-compile-upload-run cycle is also quoted by Barnes as an issue. Barnes also notes the inexact nature of a physical model’s movements and the environment in which it operates as an issue, as they can cause unpredictable actions from the robot. Simple actions such as turning through a fixed angle can be difficult to configure. Barnes argues that on balance it would be better to use the physical models to enhance and support an introductory programming course rather than the basis for a whole course.

A more practical issue raised by users of robots is the physical nature of the artefacts and the need for re-organisation of lecture rooms to facilitate their usage; cost is also an issue with using robots but this to some extent has been overcome by Linder (op. cit.) where the other robots may be shared by multiple students from different PCs.

While there is no doubt that the use of robotics in teaching programming is becoming popular, there is little empirical evidence to support it as being any more effective pedagogically that a traditional method of teaching. While most users will claim that it has improved the motivation of the students and that the students find using the robots fun, this does not make them more effective, the same question is put by Fagin et al (op. cit.), who is gathering empirical evidence to determine whether robots are an effective means of teaching programming. While there is no disputing the pedagogical benefits of collaborative active learning one must question the way in which the robots are being used in the courses, if they are just for project implementation after a series of topics have been learned traditionally, or whether, like Fagin and his colleagues, they are to be the fundamental basis of the course, and used to teach programming concepts from the outset. One would assume that by fully integrating the use of robots into the course the benefit would be more marked but without evidence to support this, we can only hypothesise.

While research has shown that PBL has proven itself effective, especially in the medical community there has not been a lot of research into its effectiveness in computer science. While there is some support for a problem based approach to teaching programming,(Greening et al, op. cit.), the findings of McCracken (op. cit.) should be noted, he found that students are “playing the game” and not in fact learning as we would like them to learn. McCracken (op. cit.) warns that the
superficial success of an intervention such as PBL can be misleading as his ethnographic study showed. He observed that deep learning and meta-cognitive skills were not achieved and the group work was not effective. However he did acknowledge the manner in which the PBL process was facilitated and indicated that it was less rigorous than the medical implementation and that the facilitators were not as expert as medical school facilitators. This would lead one to question whether the lack of success was due to the PBL process or this particular facilitation of that process. Issues raised by the Working Group on Problem Based Learning (Ellis et al, op. cit.), such as the problem of evaluation and assessment, need to be addressed if this approach is to be widely implemented. Ellis argues that general experience has shown that that assessment may threaten creative problem solving, in that it may stifle creativity. The time factor is not to be underestimated in developing appropriate problems for problem based learning courses. Where the results of a PBL trial in the University of Sydney showed no statistical difference between the conventionally taught group and the PBL group in terms of exam results, the PBL group showed more enthusiasm and positive feelings towards learning computer science. They also learned other skills, group working and independent learning skills which are essential for life long learning. By developing their own learning issues, students are expected to develop a sense of ownership of the learning process leading to greater cognitive engagement (Vat (2001)). If the PBL approach succeeds the student constructs his own knowledge by finding it himself, discussing it’s meaning with others, and refining his own knowledge as appropriate. The PBL process should promote meta-cognition through encouraging students to reflect on the problem-solving process, all factors which concur with constructivism.

Problem based learning, cognitive apprenticeship and case-based learning emphasise support to the construction of knowledge (Enkenberg, op. cit.) and these approaches with the developments of the Web, e.g. Webworlds (Chalk op. cit.), facilitate electronic collaborative work. Chalk (2001 argues that as ”regular attendance is in steep decline the use of virtual learning environments can be used to assist or scaffold learning using these Webworlds”. One problem that he encountered, which other studies confirm, was a high percentage of students (80%) felt that the tutor’s presence inhibits communication, suggesting that students will not pose questions if they think they will lose marks for appearing not to know the solution to the problem. This results in the high ability students making the most out of such environments, as is borne out in previous results. Further tests carried out by Chalk (2000a) indicate a correlation between BSCW usage and the quality of work produced, however Chalk acknowledges that this does not show that one causes the other.
The traditional role of the teacher also must move to one of facilitator or broker between the students and the experts, in the case of apprenticeship (Chalk op. cit.). This evolving role may be an issue for some teaching professionals. While there is a lack of published research on apprenticeship learning in computer science, there is anecdotal evidence to believe that this approach is being adopted in many colleges and universities. One has only to look at work placement schemes or schemes where students work as assistants on projects with experts for a specified duration and gain course credits for this work. During their time out of college the students are following a typical apprenticeship model of being coached by an expert while involved in exploratory and eventually independent work.

There is an increase in research into approaches based on constructivism with some researchers picking certain aspects of the theory and building them into their courses, reflection, collaboration, with some success. Other developments such as Grays (Gray et al, op. cit.) web based Java tool incorporates a number of constructivist elements into it.

One must bear in mind that the approaches outlined above are not mutually exclusive and they may be used together to attain the optimum results, improved learning. However, regardless of the approach that is adopted, it must have a sound underlying pedagogy and as importantly it must be integrated tightly into the course.
Conclusions

Within the academic community there is an acknowledgement that students find programming difficult. With this as a focus, researchers are actively looking for alternative approaches or complementary aids to the traditional approach of using lectures and labs for teaching programming. The research is underpinned by a sound pedagogy, i.e. variants of constructivism. However there is a lack of evidence as to how the approaches are being incorporated into the curricula and with the exception of algorithm visualization there is a lack of empirical evidence to the educational effectiveness of the approaches. This appears to be changing with empirical studies being currently conducted into the effectiveness of other approaches, such as robots and problem-based learning. The slow uptake of new developments has been a concern for researchers, and from survey results this can be attributed mainly to the time it takes the lecturing staff to learn and implement the new approach and the lack of evidence to support it’s effectiveness. This has a “chicken-and-egg” result in that if academics are reluctant to use and evaluate new developments the empirical evidence as to their effectiveness will not be generated. Many of the Working Groups encourage adoption of new approaches and promote the evaluation of their use with a view to generating the empirical evidence necessary to substantiate them.

Regardless of tool or approach adopted the following are accepted aspects of an effective approach or tool:

- Engage student through active learning
- Allow the student a certain degree of control over their learning providing progression and revision abilities
- Provision of scaffolding mechanisms, i.e. explanations, context sensitive help, cases-studies, patterns, hypertext books
- Provision of timely feedback
- Use of collaboration
- Incorporation of self-analysis mechanisms to promote metacognitive and independent learning skills
- Provision of multiple representations of concepts, textual, visual and audio

Most technology-based tools are aimed at visualizing an existing program/algorithm and few are aimed at aiding the novice programmer design the required solution/program, e.g. FLINT (Ziegler
and Crews, op. cit), this implies that more research needs to be done in this area to get the student to the stage where he understands the design of the program, independent of the language used to implement it. Using problem domains that the students understand is critical for success and poorly designed or vague problems will only add another layer of confusion.

Soloway argues that the students find it difficult to put the pieces together to solve a problem and this is borne out by Winslow (1996), who argues that psychological studies of programming conclude that novices know the syntax and semantics of individual statements but that they do not know how to combine these features into valid programs, even when they know how to solve the given problem by hand. This is an area where patterns and templates could be used to aid the novice (Wilson op. cit.).

Much work remains to be done to evaluate the effectiveness of approaches such as problem based learning, cognitive apprenticeship and the use of the Web as a framework over which to deliver computer science education, perhaps Boroni (1999) encapsulated the work to be done when he posed the following challenge for the future as “to intensify efforts in the design, development and formal testing of resources for computer science education that make effective use of the new paradigm presented by the evolving WWW and its related technologies.”
References:


Boyle T., Yazici, S., “ A Multimedia Environment for Object Oriented Design”,


Cumberbatch, S., 1999, Java Flowchart Editor, Available http://www.sbu.ac.uk/~chalkp/proj/sharron


Engström, H., 1997, JSP Editor. Available http://www.ida.his.se/~henrike/JSP/example.chtml


Harris, J., 2001, “CS1: Where is Visual Basic?”, Journal of ???


Handy Car : Information on the construction of the robot and source code. Available http://www.plattsburg.edu/csc/HC


Kay, J., Kummerfeld, B. (1998),”A Problem-based Interface Design and Programming course”, SIGCSE

LEGO, LEGO MINDSTORMS Official Site Available: http://mindstorms.lego.com


Riley, D, 1981, “Teaching problem solving in an introductory computer science class”, 12th SIGCSE Technical Symposium on Computer Science Education

Roßling G., Naps, T.L., 2002a, “Towards Intelligent Tutoring in Algorithm Visualization”

Rodger, S. 1996 “Introducing Computer Science Through Animation and Virtual Worlds”, refffff


Stasko et al, 1996 ,[Byrne, M., D., Catrambone, R., Stasko, J., T., ], ”Do Algorithm Animations Aid Learning?”, Tech. Rep. GIT-GVU-96-18, GVU Centre, Georgia Institute of Technology, Atlanta, GA,.


Smith P. & Webb G. 2000,“The efficacy of a low-level programming Visualisation Tool for teaching Programming Concepts to Novice C Programmers”


<table>
<thead>
<tr>
<th>Study</th>
<th>Independents</th>
<th>Dependents</th>
<th>Key Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price [26]</td>
<td>Debugging medium (debugging with animated view vs. debugging without animated view)</td>
<td>Debugging time, Whether bug found</td>
<td>No significant differences were found.</td>
</tr>
<tr>
<td>Nakao et al. [13]</td>
<td>Learning medium (text only vs. text-and-animation)</td>
<td>Post-test accuracy</td>
<td>No significant medium was found favoring the text-and-animation group (n=31, df=6, p=0.73).</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 4.4</td>
<td>Data set size (9, 25, or 41 elements)</td>
<td>Post-test accuracy</td>
<td>No significant differences were found</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 6</td>
<td>Order of visual presentation (Quick sort first vs. Selection sort first)</td>
<td>Post-test accuracy</td>
<td>No significant differences were found</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 6</td>
<td>Data representation style (linear vs. vertical sticks)</td>
<td>Time to take post-test, Spatial and verbal abilities not correlated with performance</td>
<td></td>
</tr>
<tr>
<td>Lawrence [16] Ch. 6</td>
<td>Level of learner involvement (study task passivity view animation vs. study text actively view by constructing own input data sets)</td>
<td>Post-test accuracy, Time to take post-test</td>
<td>Participants who viewed animations for which they constructed their own data sets scored significantly higher on post-test.</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 7</td>
<td>Representation color (color vs. black-and-white)</td>
<td>Post-test accuracy, Accuracy on a transfer task involving a new algorithm</td>
<td>Participants who viewed black-and-white animations scored significantly higher.</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 8</td>
<td>Order of median (text-first vs. animation-first)</td>
<td>Post-test accuracy</td>
<td>No significant differences detected</td>
</tr>
<tr>
<td>Lawrence [16] Ch. 9</td>
<td>Learning medium level (learner involvement (Kremer-only vs. lecture + passivity view animation vs. lecture + actively view animation by constructing own input data sets)</td>
<td>Free-response post-test accuracy</td>
<td>On free-response post-test, participants who heard lecture and actively viewed animation significantly outperformed students who only heard lecture.</td>
</tr>
<tr>
<td>Crosby &amp; Slocum [25]</td>
<td>Learning medium (lecture vs. multimedia)</td>
<td>Pre- to Post-test improvement</td>
<td>Participants who learned with multimedia significantly outperformed participants who learned through lecture.</td>
</tr>
<tr>
<td>Byun et al. [17] Study I</td>
<td>Learning medium (animation vs. no-animations)</td>
<td>Post-test accuracy, Prediction accuracy</td>
<td>Participants who viewed animations scored significantly higher than participants who did not view animation.</td>
</tr>
<tr>
<td>Byun et al. [17] Study II</td>
<td>View a different presentation (predict the next algorithm step vs. no-prediction)</td>
<td>Post-test accuracy</td>
<td>On post-test, students who were asked to predict the next algorithm step performed significantly better than participants who did not.</td>
</tr>
<tr>
<td>Gartka [20]</td>
<td>Learning medium (animation vs. no-animations) (Note: This experiment was an attempt to improve upon a portion of the Byrne, Eames, and Nakao [17] experiment)</td>
<td>Post-test accuracy</td>
<td>No significant differences were found between the two groups.</td>
</tr>
<tr>
<td>Karm et al. [33]</td>
<td>Level of learner involvement (program algorithm vs. program algo-constructed animation vs. program algo-bricks program with a decision program)</td>
<td>Programming accuracy, Post-test accuracy</td>
<td>Participants who viewed animation scored significantly higher on post-test than participants who did not view animation.</td>
</tr>
<tr>
<td>Mulholland [27]</td>
<td>Teaching medium (three textual tracers, TTM)</td>
<td>Number of problems solved (5 minute max per problem)</td>
<td>Participants who used the graphical tracer (TTM) solved significantly fewer problems than participants who used the textual tracers (Spy, TIP, TII).</td>
</tr>
<tr>
<td>Hansen et al. [14] Study I</td>
<td>Learning medium (HiVis/HyperVis vs. text only)</td>
<td>Pre-to Post-test improvement (pseudo-code, source code, and prediction)</td>
<td>Participants who learned with HiVis significantly outperformed participants who used text only.</td>
</tr>
<tr>
<td>Hansen et al. [14] Study II</td>
<td>Learning medium (HiVis/HyperVis vs. text only)</td>
<td>Same as previous</td>
<td>Participants who learned with HiVis significantly outperformed participants who used text only.</td>
</tr>
<tr>
<td>Hansen et al. [14] Study III</td>
<td>Learning medium (HiVis/HyperVis vs. cut-and-paste text + pseudo-code solving exercises)</td>
<td>Same as previous</td>
<td>No significant differences detected</td>
</tr>
<tr>
<td>Hansen et al. [14] Study IV</td>
<td>Learning medium (HiVis/HyperVis vs. lecture)</td>
<td>Same as previous</td>
<td>Participants who learned with HiVis significantly outperformed participants who viewed lecture.</td>
</tr>
<tr>
<td>Hansen et al. [14] Study V</td>
<td>Learning medium (HiVis/HyperVis vs. text + actively view animation by selecting own input data sets)</td>
<td>Same as previous</td>
<td>Participants who learned with HiVis significantly outperformed participants who read text and interacted with XForms.</td>
</tr>
<tr>
<td>Hansen et al. [14] Study VI</td>
<td>Combination of HiVis features (fall HiVis vs. HiVis without animations, traditional vs. HiVis with pseudo-code step highlighting vs. HiVis without interactive pseudo-code)</td>
<td>Same as previous</td>
<td>No significant differences detected</td>
</tr>
<tr>
<td>Hansen et al. [14] Study VII</td>
<td>Combination of HiVis views (Conceptual/Populated/Deconstructed vs. Conceptual/Populated/Deconstructed vs. Conceptual/Deconstructed/Populated)</td>
<td>Same as previous</td>
<td>Participants who interacted with Conceptual View significantly outperformed participants who did not interact with the Conceptual View.</td>
</tr>
<tr>
<td>Hansen et al. [14] Study VIII</td>
<td>Combination of HiVis views (Conceptual/Deconstructed/Populated vs. Conceptual only vs. Deconstructed only vs. Populated only)</td>
<td>Same as previous</td>
<td>Participants who had access to at least three HiVis views, or to the Detailed View, significantly outperformed the Conceptual View and Populated View groups.</td>
</tr>
<tr>
<td>Hundhausen &amp; Doig [11]</td>
<td>Level of Learner Involvement (Self-construct visualization vs. actively view pre-defined visualization)</td>
<td>Accuracy and time on tracing and programming tasks</td>
<td>No significant differences were detected.</td>
</tr>
<tr>
<td>Jurec et al. [32]</td>
<td>Interactive prediction (use animation software that enables a prediction of next algorithm step vs. no animation software with no prediction)</td>
<td>Post-test at end of 3 weekly lab sessions, Learning line</td>
<td>No significant differences were detected on post-test.</td>
</tr>
</tbody>
</table>

Table 5: Summary of controlled experiments that consider AV effectiveness (Source, Hundhausen et al. 2002)