A Study of Visualization in Introductory Programming

Jussi Kasurinen, Mika Purmonen and Uolevi Nikula

Laboratory of Information Processing
Lappeenranta University of Technology
jussi.kasurinen@lut.fi purmonen@lut.fi uolevi.nikula@lut.fi

Keywords: POP-I.A. learning to program POP-II.A. novices POP-III.D. visualization

Abstract

The teaching of fundamental programming skills is a field that extensively uses different kinds of tools to enhance learning experience. These tools come in several sizes, offering wide range of different equipment or approaches to the teaching of introductory programming curricula. At the same time, computer sciences, and programming courses in particular, suffer from high drop-out rates and falling student grades. Students lose interest on programming because of several complex models and structures have to be learned before anything visually impressive can be created. This problem is intensified by the new multimedia environments like games and applets, whereas command line programs and data algorithms have lost impact and are not considered interesting. So can visualization tools be used to increase the student motivation and create motivational tasks to promote interest towards programming?

This paper describes a project to enhance student motivation and interest towards programming in the introductory programming course by applying visualization tool to lecture demonstrations and to the course assignments. We present the results from the course and observed student reactions to introduction of a visualization application. Finally we discuss the impact of the visual demonstrations and project assignments from the motivational point of view, and present future improvement plans and observations based on the results of the development project and course outcome.

1. Introduction

The teaching of fundamental programming skills is a field that extensively uses different kinds of tools to enhance learning experience. Within the realms of tools that are aimed for realistic, or “industrial” programming experience, these tools usually focus on either helping with the source code development process, like debuggers or editors, or ease the learning of different concepts and structures with visualization and supporting content (Pears *et al.* 2007). The completeness of the tools vary also; smallest applications may include only additional libraries that must be embedded in the source code while others consist of entire programming environment offering complete compiler and debugging tools (Boada *et al.* 2004). These tools are also popular to develop; even within Finnish universities there are several projects that offer visualization tools for fundamentals of programming (Nevalainen and Sajaniemi 2006).

The visualization tools are usually used to increase motivational aspects of the courses. Usually the tools introduce animation, visual hints, sounds, and interactivity to employ several different learning styles which support the student activity. However, studies in psychology of programming indicate that programming requires extensive practice as the mental process behind program comprehension gradually evolves with experience (McKetihen *et al.* 1981). While novice programmers understand programming structures as a series of blocks or separate linear operations, experts create a mental model that combines related structures to each other (Crosby and Stelovsky 1989, McKetihen *et al.* 1981). Even if the general consensus on programming is that becoming an expert takes ten years of active training (Winslow 1996), we interpret this so that the students should practice with actual programming tasks. The experience required to become an expert, to implement programming models, or plans, are developed by designing functional code, not by passively following presentation (Robins *et al.* 2003). Because of this, learning programming concepts with visualization tools and pre-

programming methods should be used to supplement the programming courses and enable easier transition to actual programming tasks (cf. Nevalainen and Sajaniemi 2006).

Students often lose interest on programming because complex models and structures have to be learned before anything visually impressive can be created. The students can memorize the constructs, but the motivation for doing this may be wrong: technologically oriented programming – data manipulation – is not interesting and does not promote learning because simple command line outputs are not exiting (Guzdial and Soloway 2002). Just like many other courses (Rich *et al.* 2006, Reges 2006, Hermann *et al.* 2003), our introductory course on programming suffered from high drop-out rates and falling student grades (Kasurinen and Nikula, 2007a). In our case, dropping the course was a problem because it had a negative effect on the studies as a whole: if the course was failed, it prevented participation in the advanced courses the second year, and delayed the studies in general.

Our initial solution to the situation was to develop the technical infrastructure and course teaching materials (Kasurinen and Nikula, 2007a, 2007b). This lowered the dropout-rate significantly, from 65% in 2005 to 45% in 2006. After this we moved the focus on the motivational aspects of the programming courses. Our analysis on enhancing the introductory programming course supported the findings of Guzdial and Soloway (2002): students in general required more motivational and interesting tasks to keep them interested in programming. To respond to these issues, our decision was to apply a visual demonstration tool to help students understand the concepts taught in the lectures. We also wanted to offer programming exercises with visual aspects as they seemed to promote motivation and interest towards programming exercises. Our hypothesis was that the technical infrastructure revised earlier can only go so far, and after that point we should focus on the motivational aspects that enable students to exercise and try different concepts. At least in Finland, where there are no student tuition fees, the incentive for studying requires more than just possibility to participate – it also has to be interesting.

In this paper the student responses to the introduction of visual programming assignments are reported. Our recently revised introduction to programming course was supplemented with a visualization tool that was used in the lecture demonstrations and exercises. The student responses were collected with surveys measuring student interest, tool usability, and perceived difficulty of the course with visualization tool. The results indicated that the tool did get mixed reception, but altogether increased the student performance in the course, serving as a project assignment and visual teaching aid.

The rest of the paper is structured as follows: Section 2 reviews the literature and prior research on visual demonstrations and related subjects, whereas Section 3 describes the background for the current situation and introduces the applied tool. Section 4 focuses on results and student responses to the visual demonstrations and exercises, and Section 5 provides a discussion on the results. Finally, Section 6 concludes the paper with a summary and overview of development needs.

2. Related Research

From the present study point of view the two most interesting research areas are the role of motivation and the use of demonstration tools in programming education. Guzdial and Soloway (2002) claim that as the multimedia applications, videogames, and Internet in particular, have gained ground among the computer users, the traditional approaches to programming have became old-fashioned and uninteresting to students. To counter this phenomenon they suggest employing multimedia and visual programming tools in programming courses. In a later publication Forte and Guzdial (2004) introduce the term *media literacy* to illustrate the development in computer science education where non-technical aspects were given larger role in the course contents. In practical terms, the media literacy means that programming should be a universal skill for people in the information society, focusing on media aspects like photo filtering or sound editing with easily applicable tools instead of data manipulation with emphasis on practicality and efficiency.

The impact of visualization on student learning is, for example, discussed in a paper by Myller *et al.* (2007). This paper claims that the visualization is an effective learning aide as it offers external

memory for the student in tasks like learning an algorithm or structural construct. In fact, visualization in education has been defined in the *engagement taxonomy* (Naps *et al.* 2002), which classifies student engagement in learning. This taxonomy considers lecture demonstration visualization as *viewing* engagement, whereas active learning by exercising is *constructing* engagement. Additionally, a paper by Lahtinen *et al.* (2007) notes that visualization in introductory courses can be practical enough to be adopted by students voluntarily. This phenomenon leads to the conclusion that visualization tools should offer interactivity, as they are used in self-learning environments. In some cases (Barnes and Richter 2007), this concept is taken as far as using games in the introductory courses. In general, this leads to the notion that as the visualization can be combined with several different concepts, the visualization of programming or algorithms is actually a subgroup of computer science education software solutions.

Kelleher and Pausch (2005) define taxonomy for computer science education environments, where the tools are divided into two major categories. The first category tools aim at teaching programming as a discipline, i.e., *teaching systems*, and the tools in the other category aim at teaching programming as a intermediate tool to solve problems within other context, i.e., *empowering systems*. This division is based on the approach the environments take in teaching programming or enforce the understanding of cooperation between different programming constructs. For example, the teaching systems enhance means of expression by simplifying programming code or using alternative methods for code input to ease code generation. Empowering systems, on the other hand, demonstrate different possibilities to combine abstract structures to solve problems, with little regard on how these skills could be applied in industrial programming tasks. The empowering systems include also the commercial entertainment and education software.

The research for easier and more motivational novice-oriented programming environments began already in 1960's (Kelleher and Pausch 2005) when researchers built a number of different programming languages and tools with the objective of easing the initial learning phases of programming. The early research focused on making programming accessible for the average people, a focus that has not changed considerably as the programming languages have evolved and new resources on home computing have created new challenges and opportunities. Overall, the environments for computer science education aim at increasing the comprehension of different aspects and reduce the time and effort required to learn basic computer science concepts (Kelleher and Pausch 2005).

Application of visualization also has some pitfalls, which should be avoided. A paper by McGrath and Brown (2005) describes several scenarios, where visualization has not worked or even hindered the general performance. Cultural aspects, individual differences and ambiguous objects and effects may all cause unforeseen difficulties. This concern has also been an issue earlier, as an article by Whitley (1997) notifies. The visualization tools, particularly in the field of programming, have scalability concerns which may make them unfeasible in a larger context. Whitley also raises a concern over the lack of universally applicable rules in visualization styles. Rushmeyer, Dykes, Dill and Yoon (2007) also suggest that formal education should be applied to avoid misinterpretation and cater to the effectiveness and quality issues. However, their paper also acknowledges the recent generally positive development in visualization programs and documentation quality, emphasizing on the requirement of design and quality control.

Within the Kelleher and Pausch (2005) taxonomy the program visualization tools generally fall in the category of teaching systems. In this group there are two subcategories of tools (Henriksen and Kölling 2004): *microworlds*, which focus on teaching the constructs and methods via high level of abstraction and interactivity, and *direct interaction environments*, which enable programming beyond interactive structural demonstrations. The direct interaction environments aim at easing the code development process by offering supporting tools that enhance the productivity with features such as syntax highlighting and structural proofreading. The Keller and Pausch taxonomy summarizes over fifty different computer science education systems and some of the best known tools in the program visualization category are Logo with Turtle graphics, Karel the Robot, BlueJ based on Greenfoot, and Alice.

Logo is a programming language created for teaching purposes by the Bolt, Beranek and Newman (BBN) research firm at 1967 (Logo foundation 2008). The Logo was heavily influenced by Lisp, which was also used to develop the first Logo implementations. The best known aspect of Logo programming, the "Turtle Graphics", refers to a graphical representation of a program output, which was implemented in 1969. As a programming language, the Logo was designed to be syntactically simple and easy to approach, but it also implemented more advanced constructs like I/O-operations, iteration, data structures, and algorithms. However, the design is currently over thirty years old and it has little in common with the more resent programming languages like C, Java, or Python, and thus the programming language has limited practical relevance today.

Karel the Robot (Pattis *et al.* 1997) can also be characterized as a novice-oriented programming language with graphical presentations on environment with a robot figure. In Karel, the environment allows students to move a virtual robot around the world map with predefined obstacles and pick up and carry objects around. This enables the first assignments to include problem solving in terms of creating suitable command groups for different tasks like in actual programming. Unlike Logo, The Karel command base is somewhat limited, and as a programming language it is designed to be applied for only a few lectures at the beginning of the first course rather than being used as the first real programming language (Kelleher and Pausch 2005). The basic limitations also affected the Python translation of the language called Guido van Robot (Elkner 2007), which used limited, non-extendable command base and simplified syntax. This restricts the usage to be inapplicable to the complete courses, as the features cannot be extended to cover all introductory course topics.

The visual aspect of programming was a prime focus in the development of Greenfoot (Henriksen and Kölling 2004) and Alice (Carnegie Mellon University 2008, Conway *et al.* 2008) systems. Both of these systems implement visual presentation for object-oriented programming paradigm, use visual editors for objects, and give users opportunity to interact with classes with visual programming. The main objective for these systems is to promote object-oriented thinking, enhancing the student understanding on concepts like inheritance and object relations, by reducing the amount of physical programming required to complete the tasks.

3. The Problem and the Solution

3.1. The Problem

The systematic development of the fundamentals of programming course was started two years ago by only introducing a new course project to the existing fundamentals of programming course built around the C-language. During the next year the changes included moving from the C language to Python, development of a programming guide in Finnish, and revision of the examples and weekly assignments. At this time the course project was not modified but only checked for design fit with the Python language. After the first course was kept, the development efforts were focused on making the course more motivating. Namely, 20% of the students registered for the C-language course in 2005 did nothing for the course, 39% of the students doing something for the course did not complete all the compulsory assignments, and 55% did not pass the exam. After the first revision and move to Python, 16% of the students did not do anything for the course, 35% did not complete all the compulsory assignments, and 38% did not pass the exam. This improvement is also evident in the dropout survey we did to study the reasons for dropping the course, as the number 2 cause in 2005 was *too difficult a course* and in 2006 this reason had moved to number 4. Both the years the biggest reason for dropping the course was *schedule conflicts with other courses*, and the two other reasons in the top four were *lack of time* and *student laziness* in the same order both the times. Since these quantitative measures supported our qualitative observations of motivational problems, we concluded that further improvement of the course required more motivating course materials. As it happens, we had just developed Python based material for another course on computer graphics, and we started developing the idea of introducing a visual tool to improve the student motivation. The tool development was based on the following four ideas.

Context setting: In the Finnish language the word *computer* is translated to *knowledge machine* suggesting that the machine is intelligent and knows a lot. This provides quite a different basis for the first programming lecture than for, for example, an English language lecture where the computer term has its origin in mathematics and counting, and therefore provides a natural way to introduce programming through performance of simple mathematical operations rather than knowledge management. Thus our first need was to be able demonstrate that computer is actually very simple machine and every action it makes has to be specified in minute detail in a computer program.

Basic programming constructs: Once students understand that every action a computer makes needs to be programmed, the available basic constructs have to be introduced. Since most students have used computers for playing games and writing documents etc., they are likely to know all the key concepts from the practical point of view. For example, *loading* and *saving a file* are common concepts in games just like moving and firing when appropriate button is pressed (i.e., *executing commands*), and not being able to proceed to a next level before enough credits have been accumulated (i.e., *conditional execution*). So we wanted to have a familiar looking system, possibly a game, to demonstrate the use of basic constructs like commands, conditional statements, iterations, and file operations.

Coding is interesting: The use of the basic commands familiar from the games can be expected to raise interest in how to actually implement the commands in programs. Since the course was developed based on the active learning idea, we wanted to have programming tasks that would seem justified and interesting to students. Practical understanding of how a computer game can be implemented seemed like a good candidate to create interest in coding.

Interesting course project: The observations from the previous two courses suggested that many students lost the interest in the course when the project work was introduced. Thus we wanted to move from the previous C and Unix –style text filtering project to something more interesting. A move from a character based program to graphical user interface and visual feedback seemed like something worth trying.

Having identified these four central issues with the current course implementation, we started exploring the possibilities to tackle all these issues with a single tool. The other central constraints for our solution included the fact that the course had a 135 hour workload, and we wanted to keep the all the assignments in the course personal to assure that every student learned the basic programming constructs.

3.2. Developed solution

The migration to visual environment was initiated with a search of suitable visualization tool candidates, but to our surprise the offering proved limited. As our course used Python, the amount of possibilities was drastically cut, forcing us to reconsider our strategy of selecting and applying visual programming tool to the course. The most promising candidates – Greenfoot, Alice, Logo, and Karel – all came with design constraints which made them unfeasible for us (e.g., language incompability, feature constraints or programming paradigm), as our earlier revision on the same course module (Kasurinen and Nikula, 2007a, 2007b) had recently reconstructed the entire infrastructure. We did not want to cause major changes in the course contents as our recent design was successful, so our decision was to develop the visualization tool ourselves. In terms of engagement taxonomy (Naps *et al.* 2002), our objective was to create a visualization tool that would offer constructing-level engagement in weekly assignments and responding engagement in lectures, aided by the lecturer who would ask questions and present different scenarios to solve with constructs.

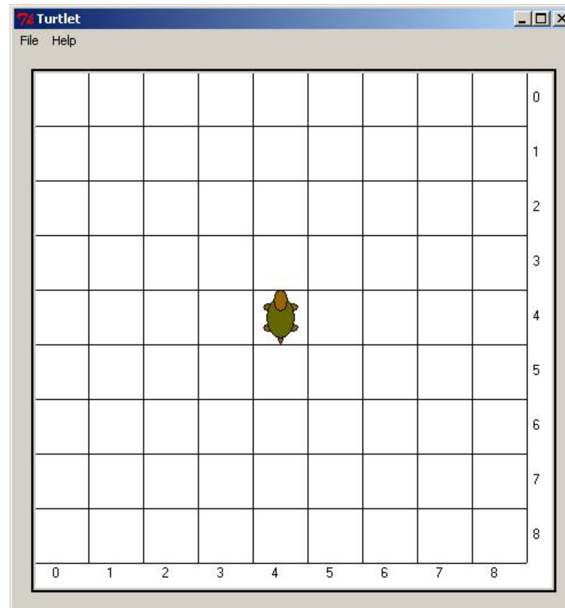

Figure 1. Map screen for the user interface

The tool was developed as a Bachelor's Thesis (Purmonen 2007) and named Turtlet after the default character in the map view. Turtlet was developed to serve two purposes, being lecture demonstration tool for programming constructs and later applied as a platform for programming assignments.

The user interface of Turtlet consists of two main windows, map screen and command screen, which are shown in Figures 1 and 2. The user interface is similar to other interactive visualization tools like Terrapin Logo or Karel the Robot as the design concept was similar. The command screen has alternatively a button interface (Figure 2) with simplified command base to allow faster and easier demonstrations in lectures. However, the Button Interface could also be used as an exercise base in GUI-exercises if the course should ever extend to cover this topic.

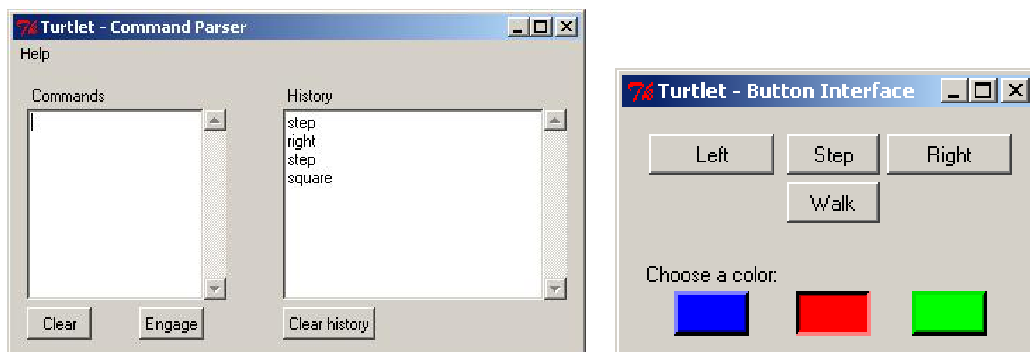

Figure 2. User interface screens; parser command screen in left, button interface for demonstrations in right

Turtlet takes commands through parser window in a command screen (the Command Parser-window in Figure 2), and shows the outcome of each command visually in the map screen. The command base can be extended with student-created commands, which was the basis for programming assignments with the tool. The students were usually given a task to implement new commands by defining Turtlet behavior for certain command words in Python programming language. An example of command *square* implementation can be seen in Figure 3.

```
import control
import visualization

control.MyCommands = ["square"]

def Parser(command):
    if command == "square":
        for i in range(0,4):
            visualization.TurnLeft()
            for j in range(0,2):
                visualization.TakeStep()
        return command
 else:
        return "Invalid command:"+command
```

Figure 3. Example code for command "square" which produces leftward two-by-two square

Students were instructed to save their own code into a single file with a predefined name so that all self-created commands were placed in this file in standard Python programming language. If command was implemented correctly, the new command would be usable in Turtlet simply by writing it to the command parser. Also, a separate system for checking the student submissions was created to automate the evaluation process and embed the exercises to a virtual learning environment.

4. Results

4.1. Using the tool in teaching

Turtlet was developed during the summer 2007 and as the lectures started in the fall, we had the first fully functional system ready. The seven first weeks of the course focused on the basic programming structures, and Turtlet was in active demonstration use in the lectures. When the course moved to more advanced topics, the Turtlet assignments were introduced to get the students to work on the course project. In these assignments, the students created and extended their own command parser in five weekly phases. After these assignments were completed, the students could start the final programming project, which required implementation of several new commands and command parameters. This final project took the last remaining two weeks of the course.

The first lecture on programming was initiated by starting Turtlet and showing the turtle in the middle of the grid. The point was that it was not doing anything before the lecturer gave a command – *Step* – and again the turtle was stuck doing nothing before another command was given. This way the fact that computer is not doing anything, unless instructed, was demonstrated to the students in practice. Another topic discussed in the first lecture was the need to follow the syntax of the commands literally. For a beginning programmer it may be hard to learn to follow the syntax precisely, and the command interface and the supporting Help-page provide one way to demonstrate the syntax issue in practice as even the lecturer needed to check the syntax from the documentation.

In the beginning of the first seven lectures focusing on the basic programming constructs Turtlet was used to demonstrate the basic concepts like iteration, conditional statements, and file operations. For example, iteration was demonstrated with the command *"Step 3"* which means taking three steps, and *"Step stop=q"* which means that the turtle takes steps until the user presses the q-key. In the code these commands are implemented as *for* and *while* statements. Similarly the conditional statement was discussed in the context of the grid boundaries – what does the turtle do when it hits the boundary? In real world a turtle cannot, of course, move past a boundary, but for computer software the possible actions are limited only by the ingenuity of the programmer. We have currently implemented, in addition to the halt, a jump to the opposite side of the grid so that the turtle can continue walking without interruptions, and the normal computer operation which means that the turtle continues a virtual walk outside the grid shown on the screen. The point of this exercise was to demonstrate the students that the computer can do almost anything in a game without real world constraints. The

commands used in the command parser are implemented in a pseudo code style meaning that they do not strictly follow any programming language per se, but look very similar to basic procedural programming language commands. The commands actually aim at the half way between natural language and the Python syntax to provide an easier transition to programming. For example, the file saving command is the following: "*Save file=TestFile.txt*."

The system has been implemented as a complete system and can be used as such in the lectures but extending the command base is also possible. The command base extension is implemented through an optional module, and every time the system is started, it checks whether the optional module is available. This optional module is the one the students are expected to develop on their own. All the modules are provided to the students in precompiled format (.pyc) except for two interface files that are provided as Python source code to enable the integration of the user-defined commands in the system. The course project is designed to have lead-in assignments as students develop the five first features of the system as weekly assignments to lower the threshold to start working on the project. The needed functions are implemented in the user created command parser (Figure 3), and the implemented commands are identified in the interface file to redirect the control module to access the student written module when necessary. As soon as the user-defined command parser is added in the system folder, the standard command parser can be removed. This way the student can choose whether he/she wants to have all the functionality available all the time during the development from two different modules, or whether he/she wants to remove the standard functionality from the system and only have his/her own functionality available.

The course project was expected to be more interesting than the previous one for two reasons. First, the students were to work on an application that looked like any other application in Windows with a graphical user interface rather than a data filter run from the command prompt. Second, developing own implementations of the Turtlet commands made it possible for the students see how the fully functional system would be crippled by removing one module, and then fixed by adding a module he/she had developed him/herself, and see how the system started to work again. Thus we hypothesized that the visualization of the commands with a graphical user interface would be more interesting and motivating to the students than the previous data centric manipulation with minimal direct feedback to the user.

4.2. Student opinions

The practical relevance of the tool was analyzed in the course by conducting two surveys on students and analyzing the survey results with additional information from the course outcome. The main objective was to determine the student reaction to the new type of exercises and project assignments, and general reaction to the visual demonstration in lectures. The student reaction to the visualization tool itself was measured with a voluntary email survey at the course week 10 after the details for course project were introduced. Additionally, the impact of visualization in general was surveyed further with another email survey, conducted right after the last lectures. The surveys consisted questions measuring attributes like difficulty, interest, and usefulness of the tool. The question on difficulty reflected on the complexity of completing assignments and understanding demonstrations, while questions on interest established how interesting and motivating the demonstrations and assignments were. Usefulness measured student-perceived benefit of both learning and exercising programming with Turtlet.

The questions in the first, tool-oriented, survey focused on different applications of visualization tool, expressing statements regarding difficulty and interest towards tool in demonstration-, exercise- and project usage. These questions also included some basic statements regarding usual software operations such as installation and user interface design. Additionally, the students were allowed to submit open comments, which were later analyzed to establish general understanding of the student opinions. 50 (35%) out of 142 registered students answered to the survey, with results reported in the Table 1.

	Positive / Yes	Neutral	Negative / No
Using Turtlet in exercises is easy	41.3%	45.7%	13%
Using Turtlet in exercises is interesting	29.5%	50%	20.5%
Turtlet-based project assignment is easy	31.9%	46.8%	21.3%
Turtlet-based project assignment is interesting	34.8%	45.7%	19.6%
Turtlet demonstrations are easy to understand	46.7%	48.9%	4.4%
Turtlet demonstrations are interesting	24.4%	42.2%	33.3%
Creating own commands with Turtlet is easy	33.3%	39.6%	27.1%
Creating own commands with Turtlet is interesting	33.4%	44.4%	22.2%
Turtlet is easy to install and operate	73.5%	16.3%	10.2%
Additional student comments	6	1	5

Table 1: Tool survey results.

In both surveys the students were asked to rate various aspects of Turtlet in scale of 1 to 5. In general, the results of the tool survey (Table 1) indicate that approximately 35 % of the students did like the tool (Positive, answers 4-5), 45% did not take any stand on either direction (Neutral, 3) and 20% of the students did not like it (Negative, 1-2). One positive finding in the tool survey was that the usability was not considered an issue: 74% said that the tool was easy to use, while only 10 % thought that interface was problematic. Only 2 students out of the 50 respondents reported that they were unable to get the tool working as instructed. The questions focusing on the visualization tool aspects indicate a cautiously positive attitude towards the programming tool. 35% of the students rated the system positive or very positive (4-5, in scale 1-5) concerning the interest as a project assignment, with 46% being indifferent (3). Similarly, 30% thought that the Turtlet assignments were interesting (4-5) with 50% being indifferent (3). 39% of the students thought that the ability to create own commands and test them with visualization was a really interesting feature. As for the usage as a visualization tool in the lectures, the tool received mixed opinions. The general consensus was that while 24 % found the demonstrations interesting or very interesting (4-5), 33 % gave negative response (1-2) and almost half (42%) did not have opinion (3). For those students, who had previous experience on programming, the exercises were perceived as easier (2.5 vs. 3.0 where 1 is easy and 5 hard), but somewhat less interesting (2.9 vs. 3.1 where 1 is boring and 5 interesting).

	Positive / Yes	Negative / No
The examples and demonstrations were easy to understand	41.3%	58.7%
The examples and demonstrations were useful	84.8%	15.2%
The course exercises in general were easy	69.3%	30.7%
The course exercises in general were useful	96.4%	3.6%
The programming project was easy	66.3%	33.7%
The programming project was useful	81.2%	18.8%
Additional student comments regarding programming exercises and/or Turtlet	13	10

Table 2: Results regarding course exercises and demonstrations from final course survey.

In the final survey (Table 2) the students were requested to answer several statements regarding the different aspects of the course like lectures, demonstrations, support literature, course assistants, and the usefulness and difficulty of the used visual tool. The main focus was on the complete course rather than just Turtlet or the programming assignments, and 82% (91 students) of the students answered it. 81% of the respondents thought that the course project with Turtlet had been useful, with 34% saying that the project was “difficult” or “moderately difficult” and 55 % “easy” or “moderately easy”. Also,

in the final survey 96% of the students thought that the programming exercises in general were useful. Also, 85% thought that demonstrations – covering both lectures and the course handbook (Kasurinen 2007) – were positive and useful for learning purposes.

The cautiously positive feelings and mixed opinions towards the tool were also present at the free comments received from conducted surveys:

Positive:

- “I have used it [Turtlet] only for the assignments, and it has worked fine. Positive feature is that you can actually see what your code is really doing…Meaning that the turtle moves.”
- “I think that it [Turtlet] is a concrete and well-suited tool for learning programming.”
- “Turtlet was interesting.”
- “I like this new system where project is completed in phases with this tool.”
- “This new project seems much more interesting [compared to the prior years].”

Negative:

- “Turtlet is boring and tedious for an engineering student. I hoped for something more practical.”
- “The project could be something that offers more room for creativity.”
- “It’s useful, and I guess it fulfills the requirements for the course project. But something more practical in the long run would have been better for me.”

In general terms, the second revision in 2007 continued to improve the overall results in the same vein the first revision in 2006 did. In 2006 our focus was to increase the number of students passing the course, i.e. the line *grade given* in Table 3, and in 2007 we tried to reduce the number of students not doing all the mandatory assignments (line *All mandatory assignments done* in Table 3). As shown in Table 3, the raise in completing all the compulsory assignments and number of grades in 2006 and 2007 was 10 percentage points. In terms of statistical significance, the course passing percentages and mandatory assignment activity relative to the amount of students from 2005 and 2006 was statistically tested with χ^2 test and found to be significant with p-value of 0.005, meaning that the results were better, not only caused by statistical variance. Between 2006 and 2007, the same test on course passing and dropout data confirmed that the course results were not statistically similar only with p-value of 0.1. In common terms this means that course 2007 was statistically better with only 90% probability, in comparison of earlier revision in 2006, when the same probability was 99.995%.

Main differentiating factor	**2005** **C**	**2006** **Python**	**2007** **Turtlet**
Some mandatory assignment done	79.5%	84.5%	88.8%
All mandatory assignments done	60.6%	64.9%	75.5%
Grade given, course passed	44.9%	62.2%	72.7%
Drop-out, withdrawal or failure total	55.1%	37.8%	27.3%
Exam failure, all other assignments done	25.8%	4.2%	3.7%

Table 3. Student participation results from courses 2005-2007.

5. Discussion

Overall, the results were something we were expecting. A ten percentage point raise in course passing and mandatory assignments was a positive outcome, aiming to the target groups we hoped for. Although the major revision was the introduction of the visualization tool as lecture demonstrations and project assignment, the minor revisions obviously also affected the results. The lecture contents

and the course manual were tuned based on the 2006 results and observations. Also the confidence level of 0.9 leaves possibility for remarks, but on the other hand, the positive comments and general interest towards to tool indicated that students liked the course project. It should also be noted that the percentage of students completing all mandatory assignments rose 10 percentage points. A comparative study with two different student groups, one using Turtlet and the other without it, was considered unrealistic goal for us for two reasons. First, our student population consists of mostly freshmen with limited programming experience but includes also senior students and individuals with extensive programming experience, so the development of two homogenous groups needed for the internal validity of the study (Yin 2003) was considered a serious challenge. And second, we did not have the resources this setup would have required. Also, several small differences in additional sources were a concern for the construct validity (Yin 2003), so the analysis focused on establishing a chain of evidence for the perceived impact of introducing visualization tool. Obviously the causal relationships of internal modules may have changed also because of other structural changes, such as changes in the course handbook or lectures. Finally, as the final results with all the implemented changes were able to produce statistically significant result with the p-value of only 0.1, it is also plausible that the impact of exclusively minor revisions would probably be insignificant.

Concerning the student feedback, the biggest unexpected outcome was the critical attitude the students had towards the new system. Although the prior courses did not use any tool of this kind, the system was technically sufficient and did not have any major flaws, the students still had mixed feelings about using it. One of the possible reasons for this could be the perceived focus and objectives of the visualization tool. It could be that some students – especially those with prior knowledge on programming – viewed the tool somewhat as a toy aimed to the complete novices and got bored with the visual demonstrations on basic operations. This possibility is supported by the fact that the amount of students with prior programming knowledge was relatively high (43%), and the visualization tool had somewhat more positive response when used as a project assignment instead of visualization tool (80% positive or indifferent versus 69%). Additionally, comparing student background records to the tool feedback, there is consistent phenomenon of experienced students thinking that the tool was easier and more importantly, less interesting. The interest towards the programming tool was 0.3 grade (in scale 0f 1-5, 5 best) lower in the exercises (2.9 vs. 3.2) and programming project (2.8. vs. 3.1), and 0.2 lower in demonstrations (2.7 vs. 2.9). As the results are homogenous in every measured course category, it seems that the tool starts to lose popularity as the programming experience is gained. Overall, the lecture demonstrations were considered the most uninteresting application of the tool even if the experienced user difference was lower than in other categories.

The validity of the surveys is also a concern; as both of the surveys were basically voluntary, the students who participated in the surveys could have been biased. For the final survey, the 82% answering percentage is obviously sufficient, but for the tool survey, the 35% answering percentage requires attention. By comparing the lecture participation average and final grade averages of the answering students to entire student body, there seems to be only minor differences between those who answered to the tool survey and those who did not. The average grade is 0.1 higher for those who answered, while lecture participation records were similar with an average of 7 participations, translating to a 50% attendance. From the students, who answered to the survey, 94% passed the course, while course average was 73%. However, it should be noted that the course average also includes those students, who dropped out from the course prior to the week 10 when the survey was conducted. Even if it seems that the students who answered the tool survey had a tendency for better odds to pass the course, the grade and attendance records indicate that the survey results represented the entire student population quite accurately rather than just a motivated subset of active students.

In general, when designing visualization tool for demonstration usage, it seems easy to create a tool for visualization, but the tool requires several other things besides graphical interface and basic setup. As the motivation on applying visualization tools to programming courses is on giving students additional support to understand the basic structures, the tool has to be interesting or at least positively accepted to get students to use it. In our case, the visualization was emphasized over content, giving students a tool that did not offer that much variability or interactivity in the end. The design further

hindered creativity as the required solutions had a tendency to be detailed and specific due to technical limitations. Some comments also suggested that at least our engineering students were somewhat disinterested towards the game like programming environment. However, it seemed that the tasks were still seen more interesting compared to the traditional small data handling exercises.

6. Conclusion

In this paper we have reported a study of improving student motivation in the first programming course with the introduction of a visual tool in the course. Our decision to improve course contents with visualization to motivate students achieved 10 percent point raise in compulsory exercises and overall passing percentages. However, the outcome indicated that there still is room for improvement.

The tool succeeded as a successor for the data manipulation project since none of the comments from re-enrolled students or others familiar with prior course projects indicated that they would have been better. The system had some technical deficiencies though, which caused some collateral difficulties: as the programming project had to clearly define the desired student command parser, some of the students felt that the project did not allow creativity in the answers, hindering the motivational aspects of the tool.

In the future, our aim is to provide extra content for the fundamental programming course assignments and try to increase the students' interest towards programming. As for the future development for the tool, the obtained results and literature (Boada *et al.* 2004, Kellerher *et al.* 2005, Robins *et al.* 2003) indicate that there are needs for visualization tools in the introductory computing curricula. In technical details, the error feedback system in general needs revision, as does the available content and exercises. The student complaints about the creativity issues should be addressed in the future. One possible direction is to develop Turtlet to better support algorithm visualization and problem solving to enable more elaborate demonstrations and assignments. These improvements should enhance the tool applicability and generate interest towards exercises and motivation for programming. As for the course, our next step could be towards collaborative programming assignments and code comprehension exercises, similarly as described in papers like Bagley and Chou (2007). These exercises would be aimed to enhance understanding of source code functionalities by proofreading and describing example source code functions, exercising topics beyond programming process like source code tracing or debugging.

7. References

Bagley, C.A. and Chou, C.C. 2007. Collaboration and the Importance for Novice in Learning Java Computer Programming. The 12th Annual Conference on Innovation and Technology in Computer Science Education, Dundee, Scotland.

Barnes, T. and Richter, H. 2007 Game2Learn: Building CS1 Learning Games for Retention. The 12th Annual Conference on Innovation and Technology in Computer Science Education, Dundee, Scotland.

Boada, I. Soler, J., Prados, F., Poch J. A teaching/learning support tool for introductory programming courses. 2004. Proc. 5th International Conference on Information Technology Based Higher Education and Training, pages 604-609, Istanbul, Turkey.

Carnegie Mellon University, Alice, www.alice.org, retrieved 7.4.2008.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., Durbin, J., Gossweiler, R., Kogi, S., Long, C., Mallory, B., Miale, S., Monkaitis, K., Patten, J., Pierce, J., Schochet, J., Staak, D., Stearns, B., Stoakley, R., Sturgill, C., Viega, J., White, J., Williams, G. and Pausch R, 2000. Alice: Lessons Learned from Building a 3D System for Novices. Conference on Human Factors in Computing Systems 2000, Hague, Netherlands.

Crosby, M. Stelovsky, J. Subject Differences in the Reading of Computer Algorithms. 1989. Designing and Using Human-Computer Interfaces and Knowledge-Based Systems, Elsevier.

Elkner, J. 2007. The Guido van Robot Programming Language, http://gvr.sourceforge.net/, retrieved 31.7.2008.

Forte, A. and Guzdial, M. 2004. Computers for communication, not calculation: media as a motivation and context for learning. Proceedings of the 37th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA.

Guido van Robot, http://gvr.sourceforge.net/, retrieved 13.3.2008.

Guzdial, M., Soloway, E. Teaching the Nintendo Generation to Program. 2002. Communications of the ACM, Vol 45(4), pages 17-21.

Henriksen, P and Kölling, M., 2004. Greenfoot: Combining Object Visualisation with Interaction. Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 73-82, Vancouver, Canada.

Herrmann, N., Popyack, J. L., Char, B., Zoski, P., Cera, C. D., Lass R. N. and Nanjappa, A. Redesigning introductory computer programming using multi-level online modules for a mixed audience. 34th SIGCSE technical symposium on computer science education. Reno, Navada, USA, ACM Press: 196-200, 2003.

Kasurinen, J. 2007. Python - programmer’s handbook, version 1.1 (in Finnish). Manuals of Department of Information Technology 10, Lappeenranta University of Technology.

Kasurinen, J. Nikula, U. 2007a. Revising The First Programming Course – The Second Round. Proc. Reflektori2007 Engineering Education Symposium, pages 92-101, Espoo, Finland.

Kasurinen, J. Nikula, U. 2007b. Lower Dropout Rates and Better Grades through Revised Course Infrastructure. Computers and Advanced Technology in Education, Beijing, China, 2007.

Kelleher, Caitlin and Pausch, Randy, 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys, Volume 37(2), pages 83 – 137.

Lahtinen, E., Järvinen, H-M., and Melakoski-Vistbacke, S. 2007. Targeting Program Visualizations. The 12th Annual Conference on Innovation and Technology in Computer Science Education, Dundee, Scotland.

McGrath, M.B. and Brown, J.R. 2005. Visual learning for science and engineering. IEEE Computer Graphics and Applications, Vol. 25(5), pages 56-63.

McKetihen, K., Reitman, J.S., Rueter H.H. and Hirtle S.C. 1981. Knowledge Organization and Skill Differences in Computer Programs. Cognitive Psychology 13, pages 307-325.

Myller, N., Laakso, M. and Korhonen, A. 2007. Analyzing Engagement Taxonomy in Collaborative Algorithm Visualization. The 12th Annual Conference on Innovation and Technology in Computer Science Education, Dundee, Scotland.

Naps, T. L., Rößling, G. R, Almstrum, V., Dann, W., Fleischer,R., Hundhausen,C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Vel´azquez-Iturbide J.A. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. In Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, pages 131–152, New York, NY, USA.

Nevalainen, S. Sajaniemi, J. An Experiment on Short-term Effects of Animated versus Static Visualization of Operation on Program Perception. Proc. 2006 International Workshop on Computing Education Research, pages 7-16, Canterbury, UK.

Pattis, R.E. Roberts, J. Stehlik, M. 1997. Karel the Robot: A Gentle Introduction to the Art of Programming, 2. edition, Wiley.

Pears A., Seidman S., Malmi L., Mannila L., Adams E., Bennedsen J., Devlin M. and Paterson J. 2007. A survey of literature on the teaching of introductory programming, ACM SIGCSE Bulletin, Volume 39(4), pages 204-223.

Purmonen, M. 2007. Development of Educational Program for Teaching of Introductory Programming (in Finnish) Bachelor's Thesis, Department of Information Technology, Lappeenranta University of Technology.

Python Software Foundation, www.python.org, retrieved 14.2.2008.

Reges S, 2006. Back to Basics in CS1 and CS2. Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, Houston, Texas, USA, 293-297.

Rich, L., Perry H. and Guzdial M., 2006. A CS1 course designed to address interests of women. 35th SIGCSE technical symposium on computer science education. Norfolk, Virginia, USA, ACM Press: 190-194.

Robins A., Rountree J., Rountree N. 2003. Learning and Teaching Programming: A Review and Discussion. Computer Science Education Vol. 13(2), 137-172.

Rushmeier, H., Dykes, J., Dill, J. and Yoon, P. 2007. Revisiting the Need for Formal Education in Visualization. IEEE Computer Graphics and Applications, Vol 27(6), pages 12-16.

Terrapin Logo Educational Software, http://www.terrapinlogo.com/, retrieved 14.2.2008.

The Logo Foundation, http://el.media.mit.edu/Logo-foundation/logo/, retrieved 14.2.2008.

Whitley, K.N. 1997. Visual Programming Languages and the Empirical Evidence For and Against. Journal of Visual Languages and Computing, Issue 8, pages 109-142.

Winslow L.E. 1996. Programming Pedagogy – A Psychological Overview. SIGCSE Bulletin, 28, 17-28.

Yin, R. K. 2003. Case Study Research Design and Methods, Third edition. Applied Social Research Methods Series Vol. 5, Sage Publications, USA.