

RAPTOR: Introducing Programming to Non-Majors with Flowcharts

Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, Steven M. Hadfield

United States Air Force Academy

Department of Computer Science

2354 Fairchild Dr, Suite 6G149

USAFA, CO 80840-6234

{Martin.Carlisle,Jeffrey.Humphries,Steven.Hadfield}@usafa.af.mil

ABSTRACT

When students are learning to develop algorithms, they very often spend more time dealing with issues of syntax than solving the problem. Additionally, the textual nature of most programming environments works against the learning style of the majority of students. RAPTOR is a flowchart-based programming environment, designed specifically to help students visualize their algorithms and avoid syntactic baggage. RAPTOR programs are created visually and executed visually by tracing the execution through the flowchart. Required syntax is kept to a minimum. Students preferred using flowcharts to express their algorithms, and were more successful creating algorithms using RAPTOR than using a traditional language or writing flowcharts without RAPTOR.

Categories and Subject Descriptors

D.1.7 [Visual Programming]

General Terms

Languages, Algorithms.

Keywords

Flowcharts, Visual Programming, Programming Environments, Problem Solving.

1. INTRODUCTION

Shackelford and LeBlanc[6] previously observed that the use of a particular programming language in an introduction to computing course tends to “annoy and distract attention from the core issue of algorithmic problem solving.” In our experience, it also distracts attention from the teaching of algorithmic problem solving. Instructors spend class time where they expect students to have the most difficulty. Consequently, they often focus on syntactic difficulties that they expect students will encounter (e.g. the inappropriate use of “=” instead of “==” in C-based languages, or the improper placement of a semicolon).

Furthermore, Felder [2] notes that most students are visual learners and that instructors tend to present information verbally. Studies [5,8] estimate that between 75% and 83% of our students are visual learners. Because of their highly textual rather than visual nature, the use of either traditional programming languages or pseudo-code provides a counter-intuitive framework for expressing algorithms to the majority of our students.

RAPTOR, the Rapid Algorithmic Prototyping Tool for Ordered Reasoning, was designed specifically to address the shortcomings of syntactic difficulties and non-visual environments. RAPTOR allows students to create algorithms by combining basic flowchart symbols. Students can then run their algorithms in the environment, either step-by-step or in continuous play mode. The environment visually displays the location of the currently executing flowchart symbol, as well as the contents of all variables. Also, RAPTOR provides a simple graphics library, based on AdaGraph [9]. Not only can the students create algorithms visually, but also the problems they solve can be visual.

We teach an “Introduction to Computing” course that is required for all students. Previously, the algorithms block of this course was taught in Ada 95 or MATLAB. This summer, we taught the same course using RAPTOR. On the final exam, we tracked three questions that required the students to develop algorithms. The students were allowed to use any method to express their algorithm (Ada, MATLAB, flowcharts, etc.) Given this choice, students preferred to use flowcharts, and those taught using RAPTOR performed better.

2. RELATED WORK

Tia Watts [10] developed SFC, a structured flowchart editor. SFC allows the user to develop a flowchart, and always displays a textual representation of the flowchart in either a C or Pascal-like syntax. The user is then required to copy and paste the textual representation into a text editor and make changes to get a complete program.

Calloni and Bagert [1] developed an iconic programming language, BACCII++, which they used as a supplement to C++ in their CS1/CS2 sequence. Their experiments showed that students using both BACCII++ and C++ performed better than those using only C++. Once the program is developed, the user can generate code for any one of five text-based languages (including Pascal and C++).

The FLINT program [3,11] eliminates the shortcoming of having to debug the textual code. Using FLINT, students create a top-down decomposition of their program, and then design flowcharts for each subgoal. These flowcharts are executed within FLINT. This forces the students into a waterfall model [7] of software engineering, whereas students may have more success with a more incremental or spiral approach.

RAPTOR has features that make it a superior teaching tool than these predecessors. As with FLINT, RAPTOR enables students to execute their algorithms within the environment, rather than having to separately compile and execute their programs. This means that debugging can be done on the visual representation of the algorithm, rather than the textual one. Unlike FLINT, RAPTOR does not force top-down decomposition, on the student, instead allowing the student to develop their code incrementally. Furthermore, RAPTOR adds one and two-dimensional arrays, files, strings and a graphics library. Students are therefore able to create more interesting programs than in the previous tools.

3. DESCRIPTION OF RAPTOR

RAPTOR is written in a combination of Ada, C# and C++, and runs in the .NET Framework. RAPTOR begins by opening a blank workspace with a start and end symbol. The user can then add flowchart symbols corresponding to loops, selections, procedure calls, assignments, inputs and outputs by selecting from the palette in the upper left corner and then inserting at an appropriate point in the flowchart (see Figure 1).

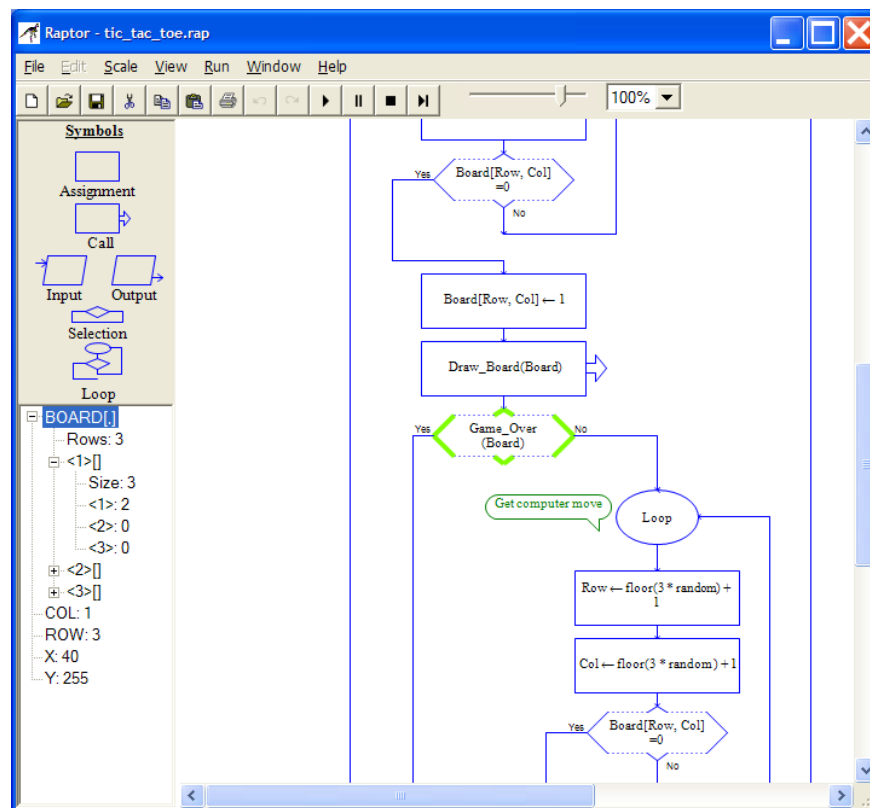


Figure 1: A RAPTOR flowchart in action.

The flowcharts are forced to be structured. Selections and loops must be properly nested, and each loop has a single exit point. Loops, however, allow the exit condition to be tested at any point inside the loop body. The student may select to use a pre-test, mid-test, or post-test loop simply by adding flowchart symbols before and/or after the loop test. Additionally, the loop structure more closely follows the loop/exit-when structure of Ada, rather than the while loop, as beginning students more naturally express positive logic (when the loop should exit) rather than negative logic (when the loop should keep going) [4].

The syntax used within a flowchart symbol is designed to be flexible. Elements have been borrowed from both C and Pascal-style languages. For example, either “*” or “^” may be used as an exponentiation operation, and “&&” or “and” may be used as a Boolean “and” operator. RAPTOR enforces syntax checking on each flowchart symbol as it is edited. Therefore, it is impossible to create a syntactically invalid flowchart. If the user enters “x+” as the right hand side of an assignment, they will get an error message and be required to fix the arithmetic expression before leaving the assignment box.

Commenting is done by right-clicking on a flowchart symbol and selecting “comment”. The comment appears as a “talking bubble” next to the flowchart symbol. The comments can be clicked and dragged to improve the aesthetic of the flowchart.

RAPTOR has over 40 built-in functions and procedures which allow the student to generate random numbers, perform trigonometric computations, draw graphics (including circles, boxes, lines, etc.), and interface with pointing devices. As seen in Figure 2, RAPTOR will automatically suggest completions to procedure names.

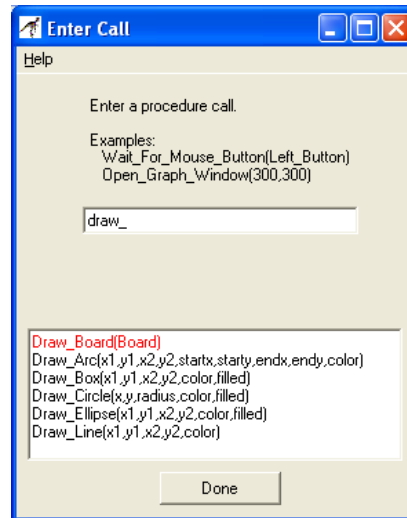


Figure 2: Entering a procedure call.

In addition, RAPTOR will automatically search the current folder for an instructor provided dynamically linked library named “plugins.dll”. If such a file is present, the student will be allowed to call those procedures from within the flowchart, and those procedures will appear in the suggestion list. This allows the instructor to create more interesting assignments by raising the level of abstraction. In Figure 2, “Draw_Board” is from the Tic-Tac-Toe sample plug-in.

During execution, the student can select to single step through the flowchart, or run continuously. The speed of execution is adjustable by moving the slider shown at the top of Figure 1. At each step, the currently executing flowchart symbol is shown in green. Additionally, the state of all of the variables is shown in a watch window at the bottom left corner of the screen.

4. EXPERIMENTAL RESULTS

The two primary goals of developing and using RAPTOR were to improve student problem solving skills while reducing the syntactical burden inherent in most programming languages. In order to initially assess the achievement of the goal of improving student problem solving skills, we compared the results of three algorithmic design questions on the final exam across the Spring 2003 and Summer 2003 offerings of our “Introduction to Computing” course, which is required of all students. The Spring 2003 offering consisted of 365 students with 15 classes using Ada and 4 classes using MATLAB. For the Summer 2003 offering, there were 16 students and RAPTOR was used as the programming language¹. Of special note is the fact that the Spring 2003 students had an average overall grade point average of 2.84 whereas the Summer 2003 offering had only an overall GPA of 2.47.

The three final exam questions used for the comparison involved a brief problem statement tasking the students to write an algorithm to solve the problem. The first question asked the students to get three numbers from the user and print the numbers starting with the first number through the second number but excluding the third number. Examples of special cases were provided for clarification. For the Spring offering, students could express their algorithms in either a flowchart, Ada, or MATLAB with about 95% choosing to use flowcharts. Only the flowchart option was provided for the Summer offering. The second question had a bowling theme testing loops and accumulators. The user would enter the scores for a team of four bowlers playing three games. The program would validate each score, re-prompt on invalid scores, and then calculate a total score for the team. The third question dealt with selection and had a Severe Acute Respiratory Syndrome (SARS) theme. The program asked an airline passenger four health-related questions; one was their temperature and the other three were yes/no questions. If the answers to two or more of the questions indicated the possibility of SARS, the program would direct the passenger for further examination otherwise it would release the passenger to board the aircraft. For both the bowling and SARS questions, the Spring offering required the solution to be expressed in either Ada or MATLAB and the Summer offering used flowcharts.

¹ The Summer 2003 offering actually included a second class of another 16 students but they were not included in the test group as their instructor had little experience with RAPTOR.

The results of the final exam questions comparison is shown in Figure 3. In all three cases, the students taught with RAPTOR performed significantly better than the students taught using Ada or MATLAB.

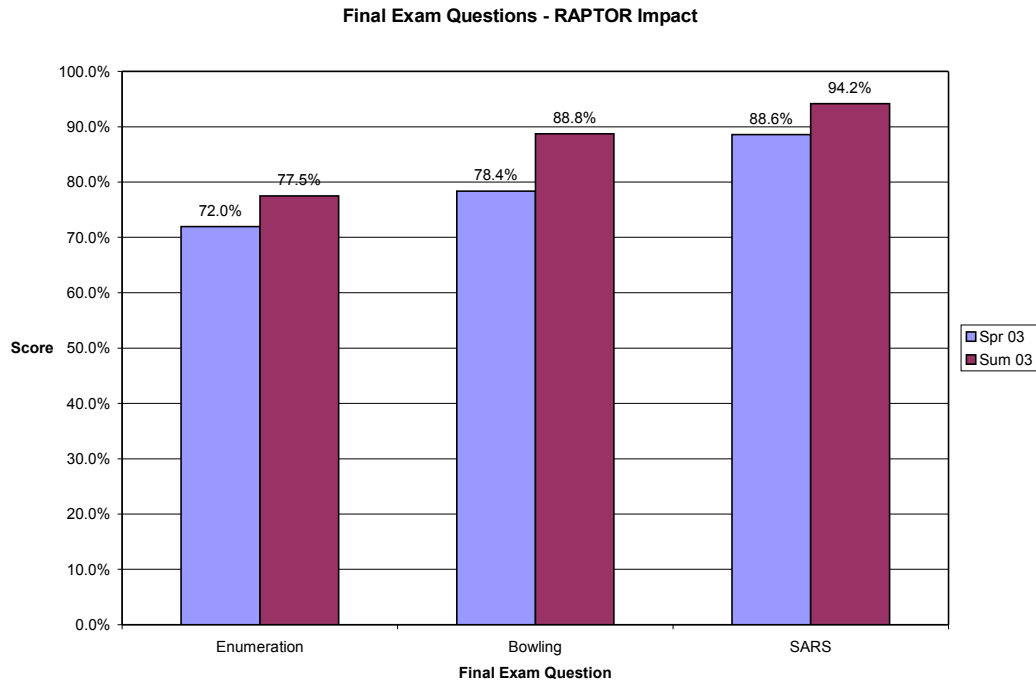


Figure 3: RAPTOR Final Exam Comparison Results

Two-sample T-tests were conducted on the results. The null hypothesis was that the mean for students using RAPTOR was less than or equal to the mean of the students using Ada or MATLAB. We constructed the null hypothesis this way as we believed that scores would improve with the use of RAPTOR. The results of these one-tailed tests are shown in Table 1 and indicate the results are statistically significant at the 82%, 95%, and 88% levels of confidence respectively which is especially impressive given the overall GPA differences of 2.84 to 2.47 from the Spring to Summer offerings.

Table 1: RAPTOR Final Exam Comparison Statistics

		Enumeration	Bowling	SARS
Spring 03	Average	72.00%	78.40%	88.60%
	Std Dev	23.40%	24.20%	18.10%
	N	365	365	365
Summer 03	Average	77.50%	88.80%	94.20%
	Std Dev	22.30%	17.50%	9.70%
	N	16	16	16
One-Tailed T-Test	S pooled	23.34%	23.92%	17.89%
	T-Stat	0.9257	1.6965	1.2136
	P-Value	0.1776	0.0453	0.1128
	Significance	82.24%	95.47%	88.72%

Also of note was an observation by the graders of the Summer offering exams that the flowcharts from this offering tended to be much more structured and hence were much easier to read and evaluate. This was attributed to the structured manner in which RAPTOR flowcharts must be constructed using the four primitives provided in the programming environment.

We do not consider these results conclusive due to the relatively small Summer 2003 population size and the facts that the Summer offering made use of a compressed schedule and was the only course the students were taking at the time. However, they are certainly promising. We plan to conduct a similar comparison between the Spring 2003 offering and the Fall 2003 offering where the population sizes will be similar as will be the other factors.

In order to assess the ease-of-use goal, we made use of a survey administered to the Summer 2003 students. The survey consisted of nine questions each with a seven-point Likert scale (1-Strongly Disagree..4-Neutral..7-Strongly Agree). Table 2 below shows the questions and the average scores.

Table 2: RAPTOR Survey Results

Survey Question	Average Response
1) I had few problems learning how to use RAPTOR to create my programs.	4.3 – Neutral to Mildly Agree
2) I had few problems getting my programs to run once I had created them.	4.4 – Neutral to Mildly Agree
3) I found the Help System in RAPTOR to be useful.	4.7 – Neutral to Mildly Agree
4) I used the Help System in RAPTOR frequently.	3.6 – Mildly Disagree to Neutral
5) RAPTOR helped me to develop and improve my problem solving skills.	5.4 - Mildly Agree to Agree
6) RAPTOR helped me to better understand how computer programs operate.	5.8 - Mildly Agree to Agree
7) I enjoyed programming in RAPTOR.	5.1 – Mildly Agree
8) Being able to view the contents of variables helped me to test and debug my programs.	6.0 – Agree
9) My teacher gave me enough instruction on using RAPTOR so that I could use it effectively to create programs.	5.7 – Mildly Agree to Agree

Questions 1 and 2 dealt directly with ease-of-use and averaged mildly positive responses. Questions 3 and 4 addressed the RAPTOR Help System which was still evolving with this initial use of RAPTOR. Questions 5 and 6 had more positive responses and indicated that the students perceived RAPTOR to be effective in helping the students develop problems skills and understanding the computer. Question 7 results indicated that the students tended to enjoy using RAPTOR which is viewed as an important result as it shows that RAPTOR tended to engage the student. This same conclusion was supported by anecdotal feedback from the students to their instructor. Question 8 results support the conclusion that variable watch features in RAPTOR were beneficial. Question 9 results were promising in that the instructor spent only twelve hours in class time with programming in RAPTOR to cover comprehensive use of the environment; sequential, conditional, and iterative control; nesting control structures, use of all the RAPTOR graphics routines, and one- and two-dimensional array manipulation. Of these twelve hours, less than two were spent on syntax and use of the environment. Unfortunately, we did not have a baseline survey from the Spring 2003 offering using Ada and MATLAB for comparison. However, we were encouraged by these results and have since implemented a number of additional ease-of-use features including most of the suggestions provided by students on the narrative portion of the survey.

5. FUTURE WORK

Our summer experiment showed very promising results using RAPTOR, although the sample size was very small. We intend to continue to track results using RAPTOR on a group of approximately 600 cadets during the Fall 2003 semester.

Additionally, we will continue to improve the error message facilities in RAPTOR, and add features to the environment.

6. CONCLUSIONS

RAPTOR provides a simple environment for students to experiment with developing algorithms. Instructors can customize the environment and facilitate more interesting exercises by adding to the built-in procedures.

Students, when given a choice, overwhelmingly prefer to express their algorithms visually using flowcharts. Even when primarily taught a third generation programming language, 95% of students chose instead to use a flowchart on the final exam. The visual nature of the flowcharts makes it easier for students to follow the control flow in their programs, and to solve problems more easily.

In a small experimental section, we found that students using RAPTOR who entered the course with a much lower incoming GPA outperformed students with a higher incoming GPA using Ada or MATLAB.

7. REFERENCES

- [1] Calloni, B. Iconic Programming Proves Effective for Teaching the First Year Programming Sequence. Proceedings of the 28th SIGCSE Symposium (1997), 262-266.
- [2] Cardellini, L. An Interview with Richard M. Felder. Journal of Science Education 3(2), (2002), 62-65.
- [3] Crews, T., and Ziegler, U. The Flowchart Interpreter for Introductory Programming Courses. Proceedings of FIE '98, 307-312.
- [4] Feldman, M., and Koffman E. Ada 95 Problem Solving and Program Design, 3rd edition. Addison-Wesley Publishing Company, 1999.
- [5] Fowler, L., Allen, M., Armarego, J., and Mackenzie, J. Learning styles and CASE tools in Software Engineering. In A. Herrmann and M.M. Kulski (eds), Flexible Futures in Tertiary Teaching. Proceedings of the 9th Annual Teaching Learning Forum, February 2000. <http://ccea.curtin.edu.au/tlf/tlf2000/fowler.html>
- [6] Shackelford, R., and LeBlanc, R. Introducing Computer Science Fundamentals Before Programming. Proceedings of FIE '97, 285-289.
- [7] Sorensen, R. A Comparison of Software Development Methodologies. Crosstalk (January 1995).
- [8] Thomas, L., Ratcliffe, M., Woodbury, J. and Jarman, E. Learning Styles and Performance in the Introductory Programming Sequence. Proceedings of the 33rd SIGCSE Symposium (March 2002), 33-42.
- [9] vanDijk, J. AdaGraph. Online. Internet. Available:
<http://users.ncrvnet.nl/gmvdijk/adagraph.html>.
- [10] Watts, T. SFC – A Structured Flow Chart Editor Version 3. Faculty Poster SIGCSE 2003. Available at:
<http://www.cs.sonoma.edu/~tiawatts/SFC/>.
- [11] Ziegler, U., and Crews, T. An Integrated Program Development Tool for Teaching and Learning How to Program. Proceedings of the 30th SIGCSE Symposium (March 1999), 276-280.