RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving

Martin C. Carlisle, Terry A. Wilson¹, 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 rather 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 visual programming environment, designed specifically to help students envision their algorithms and avoid syntactic baggage. RAPTOR programs are created visually and can be executed visually by tracing the execution through the program. Required syntax is kept to a minimum. Students preferred expressing their algorithms visually, and were more successful creating algorithms using RAPTOR than using a traditional language or writing flowcharts.

Categories and Subject Descriptors
D.1.7 [Visual Programming]

General Terms
Languages, Algorithms.

Keywords
Visual Programming, Programming Environments, Problem Solving, Flowcharts.

1. INTRODUCTION
Shackelford and LeBlanc [8] 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 [6,10] 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.

We designed RAPTOR, the Rapid Algorithmic Prototyping Tool for Ordered Reasoning, specifically to address the shortcomings of syntactic difficulties and non-visual environments. RAPTOR allows students to create algorithms by combining basic graphical 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 symbol, as well as the contents of all variables. Also, RAPTOR provides a simple graphics library, based on AdaGraph [11]. 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 twelve hour algorithms block of this course was taught in Ada 95 or MATLAB. Beginning in the summer of 2003, 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 a visual representation, and those taught using RAPTOR performed better in general.

2. RELATED WORK
Within the context of End User Development, Fischer, Giaccardi, Ye, Sutcliffe, and Mehandjiev [5] implore the benefits of graphical languages over textual ones by stating:

“Text-based languages tend to be more complex because the syntax and lexicon (terminology) must be learned from scratch, as with any human language. Consequently, languages designed specifically for end users represent the programmable world as graphical metaphors containing agents that can be instructed to behave by condition-action rules. The aim is to reduce the cognitive burden of learning by shrinking the conceptual distance between actions in the real world and programming.” [5]

Indeed, IBM has endorsed the importance of visual programming environments for end users by their use of a flowchart-based development environment within their WebSphere product [7].

¹Work performed while this author was a member of the faculty at the US Air Force Academy. Current address: ATR Technology Division AFRL/SNAR; Wright Patterson Air Force Base, Ohio; Terry.Wilson@wpafb.af.mil.

This paper is authored by an employee(s) of the United States Government and is in the public domain.

SIGCSE ’05, February 23–27, 2005, St. Louis, Missouri, USA.
ACM 1-58113-997-7/05/0002.
Tia Watts [12] 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 [3,13] 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 [9] of software engineering, whereas students may have more success with a more incremental or spiral approach.

Visual Logic [14] is a follow-on project to FLINT designed for CS 1 before transitioning to Visual Basic. It abandons the waterfall model of programming, and adds support for one-dimensional arrays and turtle graphics.

The combination of RAPTOR features makes it a superior teaching tool compared to these predecessors. 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 and prevents having to use multiple tools. RAPTOR does not force top-down decomposition, on the student, instead allowing the student to develop his or her code incrementally. Furthermore, RAPTOR adds one and two-dimensional arrays, files, strings and a more sophisticated graphics library allowing user interaction. Students are therefore able to create more interesting programs than with 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 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 program (see Figure 1).

RAPTOR programs 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 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 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 symbol as it is edited. Therefore, it is impossible to create a syntactically invalid program. 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 symbol and selecting “comment”. The comment appears as a “talking bubble” next to the symbol. The comments can be clicked and dragged to improve the aesthetic of the program.

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.

These improvements make RAPTOR a superior teaching tool compared to its predecessors.
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 RAPTOR program, 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_Bowling” is from the Tic-Tac-Toe sample plug-in.

During execution, the student can select to single step through the program, 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 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, Fall 2003, and Spring 2004 offerings of our "Introduction to Computing" course, which is required of all students. Final exams are never returned to the students, which helps avoid any effects related to question reuse. The Spring 2003 offering consisted of 365 students with 15 classes using Ada and 4 classes using MATLAB. For the Fall 2003 offering, there were 530 students and RAPTOR was used as the programming language. The Spring 2004 offering consisted of 429 students.

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 write an algorithm that would 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. RAPTOR was the only option provided for the Fall 2003 and Spring 2004 offerings. 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 body 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 2003 offering allowed the solution to be expressed in either Ada or MATLAB and the Fall 2003 and Spring 2004 offerings used RAPTOR.

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

![Figure 3: RAPTOR Final Exam Comparison Results](image)

Both one-sided and two-sided, two-sample T-tests were conducted on the results. The null hypothesis for the one sided tests was that the mean for students using RAPTOR was less than or equal to the mean of the students using Ada or MATLAB. For the two-sided tests, the null hypothesis was that the means would be equal. The results of these tests are shown in Table 1. The results indicate statistically significant increases in performance on both the Enumeration and SARS questions when using RAPTOR. The Bowling questions results were less clear cut. For Spring 2003 versus Fall 2003, there was actually a statistically significant decrease when using RAPTOR. The Spring 2003 versus Spring 2004 results showed a slight increased performance when using RAPTOR but it was far from statistically significant. We attribute this lack of increased performance with RAPTOR on the bowling questions to be due to the fact that arrays in RAPTOR are implicitly declared and hence less obvious to the students. In addition, the programming assignment for arrays during the Fall 2003 was far more challenging than the other semesters. We consider it likely that the students in the Fall 2003 semester were confused about arrays due to the complexity of the assignment and hence performed poorly on the array algorithm during the final exam.

<table>
<thead>
<tr>
<th>Question</th>
<th>Spr 03 Average</th>
<th>Fall 03 Average</th>
<th>1-Tailed Significance</th>
<th>2-Tailed Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>72.0%</td>
<td>76.3%</td>
<td>99.79%</td>
<td>99.57%</td>
</tr>
<tr>
<td>Bowling</td>
<td>78.4%</td>
<td>74.7%</td>
<td>97.44%</td>
<td>94.88%</td>
</tr>
<tr>
<td>SARS</td>
<td>88.6%</td>
<td>92.6%</td>
<td>99.96%</td>
<td>99.92%</td>
</tr>
<tr>
<td>GPA</td>
<td>2.84</td>
<td>2.93</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>Spr 03 Average</th>
<th>Spr 04 Average</th>
<th>1-Tailed Significance</th>
<th>2-Tailed Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>72.0%</td>
<td>76.0%</td>
<td>98.61%</td>
<td>97.23%</td>
</tr>
<tr>
<td>Bowling</td>
<td>78.4%</td>
<td>79.0%</td>
<td>63.74%</td>
<td>27.48%</td>
</tr>
<tr>
<td>SARS</td>
<td>88.6%</td>
<td>92.7%</td>
<td>99.97%</td>
<td>99.95%</td>
</tr>
<tr>
<td>GPA</td>
<td>2.84</td>
<td>2.87</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Also of note was an observation by the graders of the final exams that student’s algorithms from the Fall 2003 and Spring 2004 offerings 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 programs must be constructed using the six primitives provided in the programming environment.

In order to assess the ease-of-use goal, we made use of a survey administered to the Fall 2003 and Spring 2004 students. The survey consisted of nine questions each with a seven-point Likert scale (1-Strongly Disagree..4-Neutral..7-Strongly Agree). Table 2 shows the questions and the average scores.

The survey questions and the Fall 2003 and Spring 2004 results are reported on Table 2 below. The results are also shown graphically in Figure 4. All of the questions except for #7 on their enjoyment of using RAPTOR resulted in average responses that were above the neutral rating of 4.0. Students have traditionally rated the programming section as their least favorite portion of the course.

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 surveys.

<table>
<thead>
<tr>
<th>Survey Question</th>
<th>Fall 2003 Avg</th>
<th>Spring 2004 Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) I had few problems learning how to use RAPTOR to create my programs.</td>
<td>4.68</td>
<td>4.33</td>
</tr>
<tr>
<td>2) I had few problems getting my programs to run once I had created them.</td>
<td>4.56</td>
<td>4.27</td>
</tr>
<tr>
<td>3) I found the Help System in RAPTOR to be useful.</td>
<td>4.56</td>
<td>4.51</td>
</tr>
<tr>
<td>4) I used the Help System in RAPTOR frequently.</td>
<td>4.37</td>
<td>4.57</td>
</tr>
<tr>
<td>5) RAPTOR helped me to develop and improve my problem solving skills.</td>
<td>4.96</td>
<td>4.39</td>
</tr>
<tr>
<td>6) RAPTOR helped me to better understand how computer programs operate.</td>
<td>5.13</td>
<td>4.50</td>
</tr>
<tr>
<td>7) I enjoyed programming in RAPTOR.</td>
<td>3.77</td>
<td>3.40</td>
</tr>
<tr>
<td>8) Being able to view the contents of variables helped me to test and debug my programs.</td>
<td>5.42</td>
<td>4.90</td>
</tr>
<tr>
<td>9) My teacher gave me enough instruction on using RAPTOR so that I could use it effectively to create programs.</td>
<td>5.01</td>
<td>4.55</td>
</tr>
</tbody>
</table>

5. FUTURE WORK
In the upcoming semesters, we plan to further experiment with using RAPTOR to teach algorithmic programming by refining and expanding the programming assignments that we give to our students as well as enhancing the manners in which we teach our students to use RAPTOR. Of particular attention will be the issue of teaching and using arrays in RAPTOR as the final exam results indicated this to be an area for improvement. In addition, we will continue to modify and improve the RAPTOR environment with richer sets of available functions and procedures, enhanced Help facilities, and other ideas to be gleaned from user feedback.

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, overwhelming prefer to express their algorithms visually. 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 RAPTOR makes it easier for students to follow the control flow in their programs, and to solve problems more easily.

Experimental results indicate that teaching programming in RAPTOR develops problem solving skills better than teaching programming in a more traditional, non-visual language. Of note here is the observation that this conclusion did not extend to the use of arrays in problem solving. In fact, the first semester with RAPTOR students performed statistically significantly worse on the array question. This would suggest that array handling in RAPTOR might be an area for future improvements.

We have provided a web site where other universities can download RAPTOR. It is located at http://www.usafa.af.mil/dpcs/bios/mcc_html/raptor.html.
7. REFERENCES


