

Toward a More Effective Visualization Tool to Teach Novice Programmers

John C. Giordano
U.S. Military Academy
D/EECS
West Point, NY 10996
+1-845-938-3071

john.giordano@us.army.mil

Martin Carlisle
U.S. Air Force Academy
2354 Fairchild Dr, Ste 6G101
USAFA, CO 80840-6234
+1-719-333-3590

carlisle@acm.org

ABSTRACT

Algorithm visualization systems and techniques have been introduced in a number of curricula to increase learner engagement and improve pedagogical processes. Effective visualization tools must be educationally beneficial to the learner while avoiding excessive overhead. In an introductory IT course, we compared the effectiveness of a COTS diagramming tool to RAPTOR, a visual programming environment based on flowcharts. Our results indicate the COTS diagramming tool was overly heavyweight and cumbersome. Students using RAPTOR became more actively engaged in developing algorithms.

Further, we discovered that introductory IT students who had been introduced to RAPTOR performed better on the final exam than those who had only used the COTS tool. Additionally, these students overwhelmingly preferred RAPTOR to the COTS tool, and indicated that using RAPTOR made it easier for them to develop Java programs. Based on these results, we will use RAPTOR for all sections of the course starting this fall.

Categories and Subject Descriptors

K.3.2. [Computers and Education]: Computer and Information Science Education – Curriculum, Computer Science Education, Information Systems Education

General Terms

Algorithms, Design, Human Factors.

Keywords

Flowcharts, designs, RAPTOR, Java.

1. BACKGROUND

Algorithm visualization systems and techniques have been introduced in a number of curricula to increase learner engagement and improve pedagogical processes [4, 6, 7, 8, 9, 12]. At the U.S. Military Academy (USMA), we teach a core IT course that is required for all first year students. While the emphasis of the course is on IT-enabled problem-solving, much of the content is dedicated to teaching students how to design an algorithm and then implement the design in Java. In this paper, we describe how the use of automated design tools has evolved

over several semesters, resulting in the selection of a flowcharting tool that helps students become better programmers.

1.1 Course Background

In previous semesters, the development of student designs was accomplished using a simplified approach to flowcharting. In its early stages, this flowcharting methodology was not rigorously defined and students were not restricted to a single format for completing their flowcharts – they could even complete them using pencil and paper! By the Fall 2004 semester, a standard, clearly defined flowcharting methodology, along with a commercial-off-the-shelf (COTS) diagramming tool were established as primary aids to the design process. The purpose of this was to provide students with a structured approach to designing solutions and an electronic tool to help them visualize their algorithms. The COTS tool served as an adaptable design workspace which included an instructor-provided “template” of flowcharting symbols and constructs. After mastering the design methodology, students learned the fundamentals of writing programs in Java using sequencing, selection and iteration. These skills were coupled such that a student could solve a problem using a flowchart and then “translate” this design into code. Since the course emphasizes problem-solving using IT and not Computer Science fundamentals, we avoid the object-oriented capabilities of Java and focus solely on procedural programming.

1.2 Flowcharting Methodology

While the flowcharting methodology in conjunction with the COTS tool provided a well-understood framework for the students to work in, daily feedback and end-of-course critiques characterized the COTS tool as difficult to manipulate and cumbersome. Although the tool allowed students to visualize their algorithms, it resulted in an unbounded problem space and offered no direct feedback to the student regarding the efficacy of their designs. These two findings exemplify the key obstacles to successfully integrating visualization technology [10].

Despite the consistent methodology and automated tool support, some students struggled with the fundamentals of flowcharting. This became so problematic that instructors developed a presentation titled “How NOT to Flowchart”, including some humorous but telling examples of student-submitted work (see Figure 1). This example serves as an anomaly, as most students were able to grasp the fundamentals of design and apply these skills in translating flowcharts to code. In fact, the course failure rate has never exceeded 1.5% in any of the past five semesters.

This paper is authored by an employee(s) of the United States Government and is in the public domain.
SIGITE'06, October 19–21, 2006, Minneapolis, Minnesota, USA.
ACM 1-59593-521-5/06/0010.

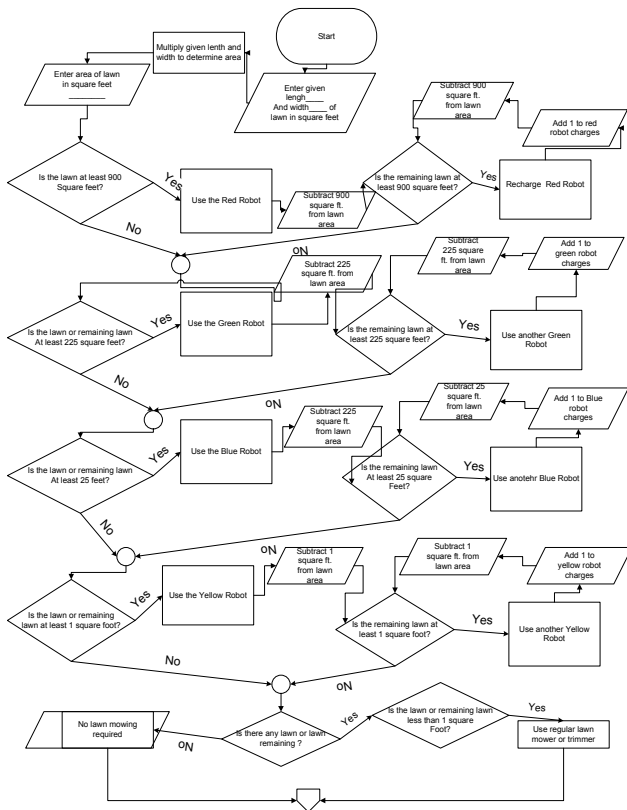


Figure 1. A Poorly Designed Student Submission

A desire to lower the failure rate along with student perceptions of the automated tools used in class precipitated a change in design paradigms.

1.3 Motivating Change

Despite the students' overall success in achieving course goals, the faculty determined the need to search for a better design platform that would eliminate the encumbrances of the COTS tool. As such, we outlined several requirements that would define the ideal replacement, including:

- Visual algorithm representation, using symbology as similar to ours as possible
- Use of syntax that is Java-like or complementary to Java
- Facilitate the transition from flowcharting to programming

While cost of ownership, maintenance and ease of use were practical concerns, we felt that identification of suitable replacement would be of benefit to the community. We were aware that several algorithm visualization and flowcharting programs had been developed for use in the classroom. Finding one that fit our needs or was appropriate for adaptation might demonstrate the flexibility of existing tools and eliminate the need for a custom application. In the following section we offer an overview of the use of visualization in introductory computer

courses, discuss our survey of the literature for an acceptable suite of replacements to consider and describe the selection of a flowcharting program to be used in a pilot study. We go on to describe the framework of the pilot study in Section 3 and report the results and findings, both qualitative and quantitative, in Section 4. In Section 5, we show how this work could be applied in similar settings and describe our continued effort to study the effects of the selected tool. Finally, In Section 6, we offer concluding comments.

2. THE ROLE OF VISUALIZATION

While algorithms are a timeless notion, the pedagogy of automated algorithm visualization emerged less than three decades ago [1, 2]. In considering the role that visualization should play in computer science education, Naps and colleagues suggest that the observed effectiveness of visualization tools in the classroom may not correlate to the intuitive expectations of instructors. They also warn of the cognitive overhead that may be born by students and instructors alike in adopting new visualization technologies [10]. In accordance with these findings, the replacement of an existing tool with one that is better suited for the purpose of algorithm visualization was expected to have little impact on instructors. Moreover, we identified a program that was easier for novice students to use. This is supported by qualitative results discussed in Section 4.

Existing course goals suggested that we should retain a high-level language as a platform for teaching, and that we should continue to introduce the syntax of Java structures with some visualization system. Consideration was given to abandoning the baggage of syntax inherent in high-level languages. Cooper and colleagues have been successful in using ALICE, a completely graphical design and programming environment to teach objects first in introductory IT & CS courses [8]. Implicit organizational goals, such as preparing students to acquire further high-level programming skills, eclipsed the desire to teach fundamentals in the most straightforward manner. The appropriate blend of visualization and high level language emerged and we decided to search for tools that could foster the supporting role of visualization. Ideally, this tool would lead to better high-level language programmers. Visualization, while critical, assumed a secondary role to syntactic and semantic correctness in Java programming.

2.1 Survey of Visualization Tools

The literature suggested that there are several well-known visualization tools offering varying degrees of functionality. The two broad categories of visualization tools include those that feature iconic programming and expose students to some form of code or pseudocode, and those that hide any form of code from the student. In this context, iconic programming may include the use of symbols, visual structures or flowcharts, all meant to represent some lower-level syntactic process that is abstracted for the user.

Calloni and Bagert introduced BACCII in 1993 [3] and BACCII++ in 1997 [4]. Both of these platforms fall into the former category and expose students to several high-level languages. BACCII generates syntactically correct code in PASCAL, C, FORTRAN or BASIC after a student constructs a

visual representation of their algorithm. BACCII++ combines a similar visual approach and generates C++ code.

Likewise, Tia Watts created the Structured Flow Chart (SFC) Editor [12]. The SFC Editor focuses solely on flowchart development and generates either a generic or C++ style pseudocode. This pseudocode cannot be modified or edited; students manipulate only flowcharts and symbols.

Cilliers, Calitz and Greyling go a step further in B#, an iconic programming notation that allows students to generate syntactically correct PASCAL code in an IDE that packages both the flowchart and code together [7].

Hundhausen and Brown combine direct manipulation of program code and program objects in ALVIS Live! [9]. The program code in this case is a custom version of pseudocode called SALSA. Students may type code directly into a Script Editor pane, or manipulate data structures and other objects in an adjacent Animation pane.

Abandoning distinct notions of code and symbols, Carlisle et al. developed RAPTOR, a visual programming environment that unifies syntax and symbology [6]. This flowcharting tool requires syntactical entries into flowcharting symbols, but prevents students from creating syntactically incorrect flowcharts. There is no direct manipulation of traditional text-based code in RAPTOR, but the syntax is Java-like.

Finally, Cooper, Dann and Pausch have developed ALICE, a visual programming environment based on 3D characters, objects and animations [8]. While these objects require some syntactic understanding, students are not required to manually enter any text in order to demonstrate understanding of logical processes.

We find that these tools span the categories of visualization paradigms for novice programmers. From tools like B# that allow the student to manipulate both symbols and code to ALICE, which rejects test-based programming altogether, there is a rich set of visualization programs and platforms to select from.

2.2 Selecting an Appropriate Tool

All of the tools considered have been implemented in at least one university-level course and have appeared in peer-reviewed literature. Those that were publicly available were acquired and reviewed by the faculty. Consideration was given to the requirements articulated in Section 1.3. Some of the implicit goals were to minimize disruption to the curriculum when implementing a replacement flowcharting tool and to complement the existing course content as much as possible. While no perfect replacement for the COTS tool emerged, the faculty considered RAPTOR to be the most adequate replacement for our purposes. Some of the features and capabilities that made RAPTOR most advantageous were:

- Standard symbol template visible to user
- Constrained symbol placement
- Input and output handled automatically
- Real-time syntax checking
- Flowcharts actually “run”; one may run a flowchart to completion or step through each symbol discretely
- Real-time variable inspection at run-time

Unfortunately, RAPTOR featured different flowcharting symbols than our methodology. Moreover, RAPTOR serves as a primary introductory programming platform and is not intended to lead to text-based programming in a single semester. Despite these differences, we contacted RAPTOR’s developers to determine if acquiring a modified version of RAPTOR would be feasible. Once we agreed on a set of requirements to bring RAPTOR more in line with our pedagogy and flowcharting methodology, a modified version of RAPTOR was produced to fit our needs at no cost. This cooperative effort demonstrates the flexibility of tools developed outside of the commercial sphere for a specific academic purpose.

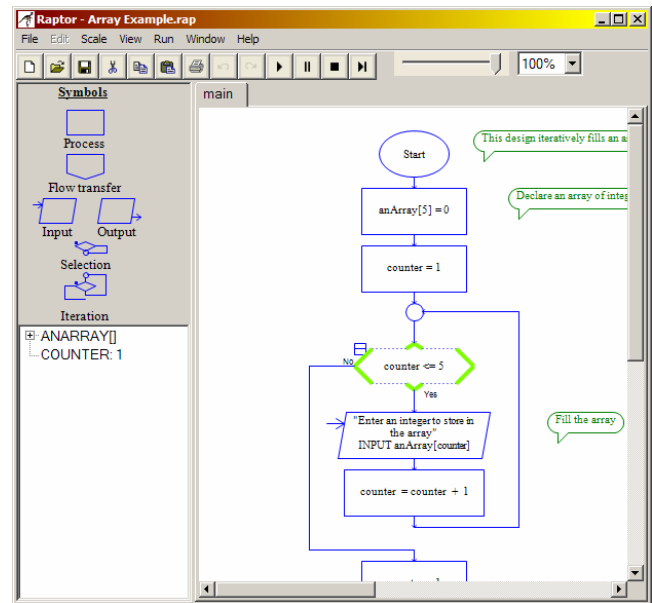


Figure 2. Customized Version of RAPTOR

After acquiring a “USMA” version of RAPTOR (Figure 2), we considered how best to integrate this program into our teaching for the Fall semester of 2005 and Spring semester of 2006. Minimally, we expect students to approach level 3 of Bloom’s taxonomy of understanding; that is, the learner would be able to apply the learned material in specifically described new situations [1]. At best, we hope to approach level 4 of the taxonomy, where the learner can identify the components of a complex problem and break them down into smaller parts. As our course was in a transition from being based on out-of-class projects to in-class closed labs, assessments in the Fall 2005 semester were going to focus on these two levels of Bloom’s taxonomy [1]. Previously, course events consisted of out-of-class projects based on complex and contextually-dependent problems, each requiring about 6-8 hours to complete. Three projects were completed individually and another was completed by two-person teams. These assignments emphasized higher levels of Bloom’s taxonomy. Small, well-scoped problems that could be solved by an individual in a two hour lab period now constitute almost half of the course points.

As a result of this transition, we sought to minimize the degree of change between the Spring 2005 and Fall 2005 semesters. Rather than integrate RAPTOR into all 32 sections of our course, we chose to conduct a pilot study involving our four advanced sections. Three of these sections (54 students) would use RAPTOR throughout the semester and the fourth section (18 students) would serve as a control group.

3. PILOT STUDY INVOLVING RAPTOR

3.1 Pilot Study Basics

After assessing RAPTOR for its capabilities, we modified some course content, assessment problems and assignments to reflect changes induced by using this new tool. Even though the basis of assessments was changing markedly from Spring 2005 to Fall 2005, much of the course content was consistent. We decided to conduct the pilot study within the context of the Fall 2005 semester. Since the course culminates in a final exam that requires both flowcharting and Java programming, we selected these as objective measures of performance. Another factor to consider was mid-semester resectioning. After about 12 lessons, the advanced sections identify students that are struggling. Likewise, students who are doing exceptionally well in the regular sections are identified. These two populations are swapped between regular and advanced sections. This results in a nearly 50% turnover in the advanced sections at mid-semester. While this turbulence may seem disruptive to both the student and the instructor, we have found it effective in keeping gifted students engaged in the challenges of the advanced section. Relocating weaker students to a more conducive environment allows them to move at a slower pace with easier problems, yet still attain course goals.

3.2 Conduct of the Course

Near the beginning of the semester, students were introduced to the standard flowcharting methodology in a platform-independent manner. As the core concepts of sequencing, selection and iteration were covered in the classroom, students in the RAPTOR treatment group used nothing but RAPTOR to create their flowcharts. Likewise, all other students were taught how to create flowcharts using the COTS tool. After design principles and tools were introduced, a web site design and coding project was required. This project required that all students, even those in the RAPTOR treatment group, complete a web map diagram of their web site using the COTS tool. This was done to ensure that students who might be resectioned out of the treatment group would have at least minimal familiarization with the COTS tool before arriving in one of the regular sections. Resectioning was completed after these design fundamentals and web programming lessons. As students arrived in their new sections, the curriculum called for the introduction of Java programming to implement the logical designs introduced earlier in the course. This allowed instructors to assist students who were resectioned to adjust to the design tool being used in their new class.

After introducing Java, course lessons required that students complete increasingly complex designs with their designated design tools, and use the flowcharts as a template for writing Java programs. Ultimately, these skills were tested during individual

in-class labs. Each lab required the student to create and submit two deliverable components (see Table 1).

Table 1. In-Class Lab Requirements

	Problem Specification	Flowchart	Java Program	Test Plan
Sequencing			X	X
Selection	X		X	
Iteration		X	X	

The course culminated with a final exam at the end of the semester. Students were tested on web design and programming, logical design, Java programming and testing principles of designs and programs. At the completion of the semester, students in the RAPTOR treatment group completed a survey regarding the design tools used throughout the course. About 50% of the treatment group used the COTS tool superficially. The remainder were introduced to the COTS tool in the regular sections, and then switched to RAPTOR after resectioning. As discussed earlier, one advanced section served as the control group and used nothing but the COTS tool. We studied the results of the treatment and control groups on three questions from the final exam – a flowcharting problem and two Java programming problems. Also, we examined the results of the treatment group’s survey. Our results and findings are presented in the next section.

4. RESULTS AND FINDINGS

In order to support a decision to implement RAPTOR fully throughout the course, we rely on both qualitative and quantitative data. Although intuitively most instructors felt that RAPTOR was a superior tool and should be used throughout the course immediately, we undertook the pilot study in order to evaluate its effectiveness objectively. We find that students exposed to both the COTS tool and RAPTOR overwhelmingly prefer RAPTOR as a platform for visually developing their algorithms. An initial concern was that using RAPTOR would undermine our efforts to teach design and flowcharting as a prelude to high-level language programming. This concern has been dispelled by the survey of the treatment group.

Despite a noted improvement in scores on final exam questions among the RAPTOR treatment group, the quantitative analysis proved statistically insignificant for these results. Although the RAPTOR sections’ averages were higher on all 3 questions, the small sample size prevented these differences from being statistically significant. Also, the Java programs were evaluated by automated means. That is, each program is tested for output against prescribed expected results. Most often, a student’s program either passed or failed all tests, resulting in a bimodal distribution of grades. In light of these findings, we continued the pilot study into the Spring 2006 semester.

4.1 Qualitative Results

The most striking result culled from the treatment group survey was that of 41 respondents, only one preferred the COTS tool over RAPTOR as a design tool. This was just one of eight questions that students were asked at the end of the course. This

was not a surprise, since students in the treatment group would often express their preference for using RAPTOR over the COTS tool as they progressed in their work.

As seen in Figure 3, most students evaluated RAPTOR as either effective or highly effective as a design tool. Furthermore, Figure 4 shows that students felt that RAPTOR made programming easier for them.

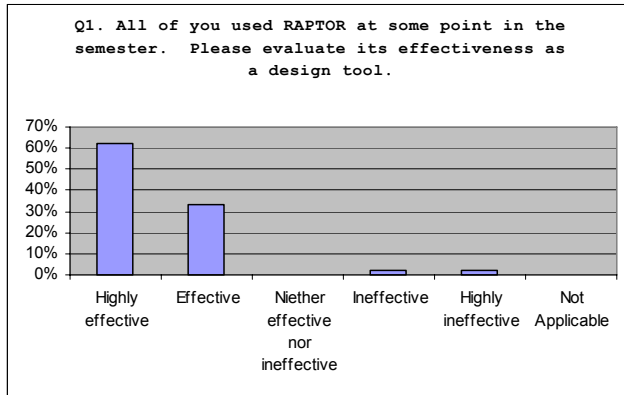


Figure 3. Survey Question 1 Results

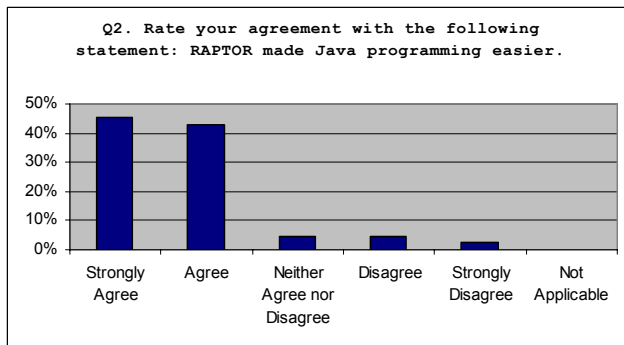


Figure 4. Survey Question 2 Results

Figure 5 illustrates that students would rather write a program when simply given a design, as opposed to conducting their own analysis and just creating the design without writing the program. This supports our pedagogical approach to separate the notions of design and coding.

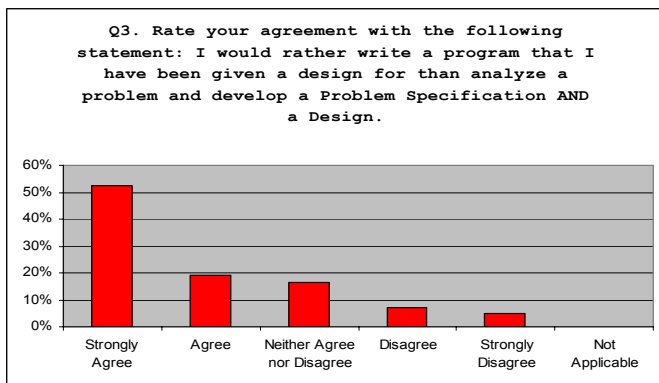


Figure 5. Survey Question 3 Results.

Figures 6 and 7 highlight the complementary nature of RAPTOR and our course content. Although we teach a modified waterfall design methodology, we found that students would often reject the design and flowcharting and go straight to coding when permitted. RAPTOR helps to reinforce a design methodology with surprising acceptance among advanced students. It also bolsters the notion that scrupulous design and flowcharting make the programming process easier. Lastly, Figure 8 demonstrates that our concern about RAPTOR undermining high-level language programming was unfounded.

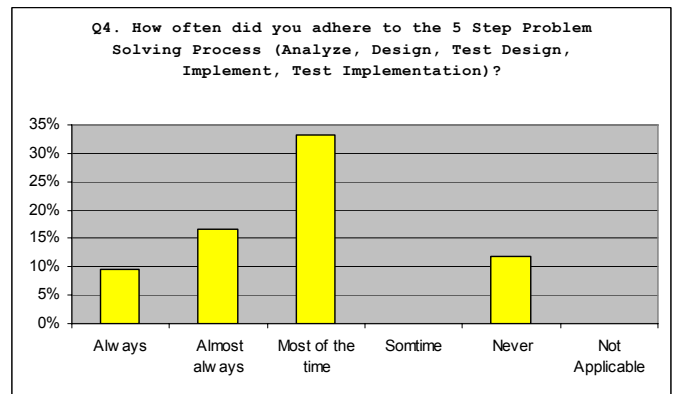


Figure 6. Survey Question 4 Results

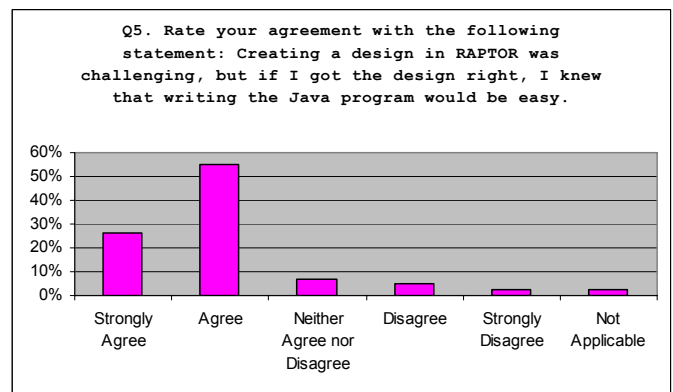


Figure 7. Survey Question 5 Results

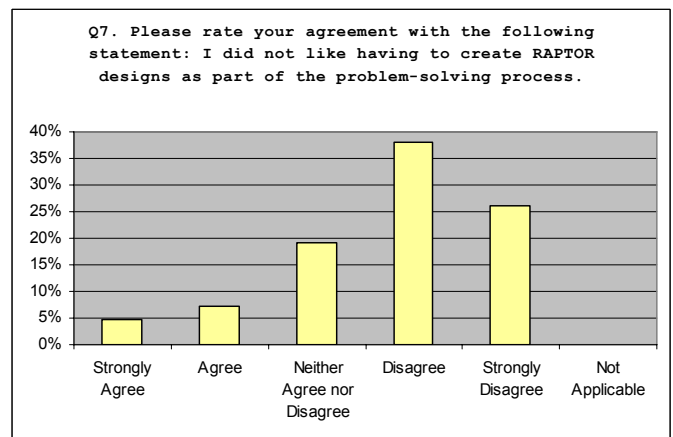


Figure 8. Survey Question 6 Results

We conclude from this that RAPTOR is not only widely accepted and preferred by students, but that it also supports course goals and enhances how students feel about programming. Although this is a subjective assessment, it is important to know that students walk away from our core course with a favorable impression of automated tools and how they can aid in learning.

4.2 Quantitative Results

Our quantitative findings are encouraging even though they are somewhat less clear than our qualitative findings. While statistically insignificant, we find that there is a marked, if not dramatic, improvement in scores in the RAPTOR treatment group. Both the treatment and the control groups demonstrate appreciably better results in the final exam questions we are concerned with. Table 2 shows the composite scores of the advanced section (treatment and control) and the regular sections (baseline).

Table 2. Final Exam Statistics from Fall 2005

	RAPTOR Treatment	COTS Control	Course Baseline
Flowchart	91.17%	85.19%	74.21%
Java 1	98.32%	97.22%	89.82%
Java 2	95.21%	87.5%	83.10%

4.3 Findings

Data from the Spring 2006 semester is not yet available, but based on the results of the Fall 2005 survey and final exam, we have decided to replace the COTS tool with RAPTOR in all sections in the Fall of 2006.

5. FUTURE WORK

In the future, we would like to offer our students a suite of visualization tools and allow them to select among several in order to best serve a highly diverse group. It would also prove interesting to investigate the effectiveness of the different visualization tools mentioned in Section 2.1 in promoting student learning. Even though the final exam changes between academic years, we would like to correlate COTS tool results from the past to those achieved by a student body using RAPTOR for flowcharting. We will continue to collaborate with the RAPTOR developers to refine our customized version, suggest improvements and seek to find ways to integrate RAPTOR into other courses where it makes sense. We encourage others in the community to consider RAPTOR, or another visualization platform discussed in the paper, as a design platform in a course that teaches logical processes or novice programming. While we have demonstrated that RAPTOR can enhance the teaching of text-based programming, its developers are comfortable with RAPTOR as a visual language and primary programming platform for novices. We are interested to see how RAPTOR can be employed in different institutions.

6. CONCLUSIONS

While the survey of visualization tools in Section 2.1 is not exhaustive, we find that what we have identified and considered as a replacement for our COTS tool represent the spectrum of programs available for such purpose. Our search for a tool that

was more lightweight, less cumbersome and better suited to the specific function of algorithm visualization has illuminated our pedagogy and improved student impressions of the programs used to support their learning. Our experimental results showed that students using RAPTOR performed better (although the sample size was too small to achieve statistical significance). These quantitative results provide the practical basis for replacing the COTS tool with RAPTOR. They demonstrate that RAPTOR positively impacts student learning and enhances learning outcomes in a discernable way. Furthermore, students greatly preferred RAPTOR to our COTS tool. This increased level of user acceptance bolsters our decision to fully implement RAPTOR throughout our course. RAPTOR may not be right for everyone and every purpose, but it is ideally suited for teaching novices how to design algorithms and write programs in a high-level language like Java.

7. REFERENCES

- [1] Bloom, B.S. and Krathwohl, D.R. *Taxonomy of Educational Objectives; the Classification of Educational Goals, Handbook I: Cognitive Domain*. Addison Wesley, 1956.
- [2] Brown, H.M. *Algorithm Animation*. MIT Press, Cambridge, Massachusetts, 1988.
- [3] Brown, M.H. and Sedgewick, R. A System for Algorithm Animation Structures. *ACM SIGGRAPH '84 Proceedings*. (Minneapolis, Minnesota, Jul 23-27, 1984). ACM Press, New York, NY, 1984, 177-186.
- [4] Calloni, B. and Bagert, D., Iconic programming in BACCII vs. textual programming. In *Proceedings of ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '94)*(Phoenix, Arizona, Mar. 1994) ACM Press, New York, NY, 1994, 188-192.
- [5] Calloni, B., and Bagert, D. and Iconic programming proves effective for teaching first year programming sequence. In *Proceedings of ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '94)*(San Jose, California, Feb. 27 – Mar. 1, 1994), 262-266.
- [6] Carlisle, M., Wilson, T., Humphries, S. and Hadfield, S. RAPTOR: a visual programming environment for teaching algorithmic problem solving. In *Proceedings of ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*(St. Louis, Missouri, Feb. 23-27, 2005). ACM Press, New York, NY, 2005, 176-180.
- [7] Cilliers, C., Calitz, A., Greyling, J. The effect of integrating an iconic programming notation into CS1. In *Proceedings of the 10th Annual SIGCSE conference on innovation and technology in computer science education (ITiCSE '05)*(Monte de Caparica, Portugal, June 27-29, 2005). ACM Press, New York, NY, 2005, 108-112.
- [8] Cooper, S., Dann, W., and Pausch, R. Teaching objects-first in introductory computer science. In *Proceedings of ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*(St. Louis, Missouri, Feb. 23-27, 2005). ACM Press, New York, NY, 2005, 191-195.
- [9] Hundhausen, C., and Brown, J. L. What you see is what you code: a radically dynamic algorithm visualization development model for novice learners. In *Proceedings of the 2005 IEEE Symposium on Visual Language and Human-*

Centered Computing (VL/HCC '05) (Dallas, Texas, Sep. 20-24, 2005) IEEE Computer Society, Los Alamitos, CA, 2005, 163-170.

- [10] Naps, T., Robling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velazquez-Iturbide, J.A. *Working Group Report from ITiCSE on Innovation and Technology in*

Computer Education (ITiCSE WGR '02)(Aarhus, Denmark, June 24-28, 2002) ACM Press, New York, NY, 2002, 131-152.

- [11] Watts, T. The SFC editor: a graphical tool for algorithm development. *Journal of Computing Sciences in Colleges*, 20, 2 (Dec. 2004), 73-85.