Journal of Agricultural Systems, Technology, and Management

Volume 23
2012

Teaching Control Programming and Problem-Solving Techniques with Karel++

Dennis G. Watson[1], Tony V. Harrison[2]

[1] Associate Professor, Department of Plant, Soil and Agricultural Systems, Southern Illinois University, Carbondale, IL 62901. Phone: 618-453-6979 Email: dwatson@siu.edu

[2] JPT Integrated Solutions, Inc. Gainesville, FL 32609

Abstract

Introductory computer programming was included in a required course for agricultural systems students. The Karel++ programming environment was selected to teach introductory programming, due to its robot programming paradigm that could be adapted for agricultural applications and the ease of which students could visualize the results of their programming efforts. Problem analysis and problem solving skills were emphasized. The Karel++ environment, program, textbook, course lectures, programming assignments, and student feedback are described.

***Keywords*:** programming, problem solving, problem analysis, control programming

Introduction

Problem solving skills are a workplace requirement for agricultural systems (AS) graduates. Two foundation applications for agricultural technology programs are “apply principles of problem solving to be able to solve problems” and “use computer technologies to solve problems” (Harper, Buriak, and Hitchings, 2001). As workplace systems, such as automation, become increasingly pervasive and complex, AS students would benefit from additional training and experience in problem solving. In 2002, we considered this need when reviewing course requirements for our AS students.

One consideration was to make computer programming a requirement in the curriculum. Many science and engineering programs require a computer programming course and there are good justifications. Programming reveals the mechanical nature of computers and the notion of computer intelligence evaporates when students learn they can control each step of a machine’s behavior (Biermann, 1994). We were concerned that AS graduates might view automation or decision making systems as a “black box.” Introductory computer programming experience would provide students with a basic understanding of control structures and the use of input and outputs. The result would be increased ability to interact with system designers and process engineers. A programming course should include general problem solving methods (Gries, 1974). A common justification for teaching programming is that it is a “study in clear thinking and problem solving, and it gives students wonderful practice at building representations and working with divide-and-conquer top-down methodologies” (Biermann, 1994). These techniques for breaking a complex problem into increasingly smaller, readily understood components transcend computer programming and are valuable strategies for solving many systems problems.

Although a computer programming course would offer some important advantages for our students, we had reservations about our students taking an available introduction to programming course. Based on descriptions of students who had completed the course, a large part of the course focused on constructs and features of the language before applications were developed, and the applications were related to business processing rather than agriculture. A course was needed that adhered to Biermann’s first rule of teaching programming to nonmajors, “that the study should focus on interesting applications” (Biermann, 1994). An ideal course would focus on applications from the very beginning and for AS students this would include agricultural or other easily relatable applications.

We considered developing a new course within our major to teach introductory computer programming with agricultural applications, but could not find a suitable text. After discussions with numerous university faculty, a community college instructor told us about Karel the Robot (Pattis, 1995). Karel is a mini programming language that has achieved widespread use (Pane and Myers, 1996; Kelleher and Pausch, 2005). Karel primarily consists of a stick-figure robot on a two dimensional plane that can be programmed to move, pick up beepers, and place beepers. Karel is a teaching tool that visually demonstrates the result of each step in a computer program, in a way that is less abstract than many programming environments.

The popularity of Karel has resulted in several descendants, including Karel++ (Bergin, Stehlik, Roberts, and Pattis, 1997), Karel J. Robot (Bergin, Stehlik, Roberts and Pattis, 2005), JKarelRobot (Buck and Stucki, 2001), and Alice (Cooper, Dann, and Pausch, 2000). Karel++, an object-oriented, C++-like version of Karel, reduced the different aspects of object-oriented programming from code, execution, model, files, and display to simply code and display (Berge, Borge, Fjuk, Kaasboll, and Samuelsen, 2003). Reported advantages of Karel++ included visual

representation of robots, fewer questions from students, emphasis on fundamentals of objects from the beginning, and students enjoyed directing robots (Becker, 2001).

Karel++ was selected for teaching introductory programming to our AS students in a three credit hour, semester-length, required course. The course was entitled, “Introduction to Control Programming.” Karel++ was used for the first seven to eight weeks of the course, compared to only four weeks in an introductory computer science course (Becker, 2001). The rest of our course focused on development of monitoring and control applications using a programmable automation controller.

This article provides a summary of the Karel++ unit designed to introduce computer programming to AS students, including a description of the Karel++ environment, Karel++ program, textbook, course lectures, programming (lab) assignments, and student feedback.

The Karel++ Environment

Karel++ inhabits a two dimensional world consisting of north-south avenues and east-west streets (see Figure 1). Both are numbered from one to infinity, starting at the origin. A world can be created with walls to block robot movement and beepers placed on intersections of streets and avenues. The robot starts by facing north, south, east, or west and can move from one intersection to an adjacent one, place or pickup beepers, make a 90° left turn, and turn off. If a robot attempts to move into a wall or pick up a beeper where none exist, it will automatically turn off. Based on these primitive instructions, in a built-in class, a new class of robot and instructions can be written to complete a task. For example, a new turn right instruction would consist of three turn left instructions. A second built-in class adds Boolean instructions so a robot can determine if the front is clear (no wall), if next to a beeper, if next to another robot, if any beepers in the robot’s beeper bag, and if facing north, south, east, or west. Students were challenged to solve a wide range of problems by breaking them down into one or more of these primitive instructions.

The Karel++ environment of street and avenue intersections and beepers can be adapted to agricultural applications. For example, the street and avenue numbers can be used as a type of GPS coordinates, beepers can be plants or animals, and the walls can be road or field boundaries. Rather than placing or picking up beepers, problems can be described as planting or harvesting plants or moving animals.

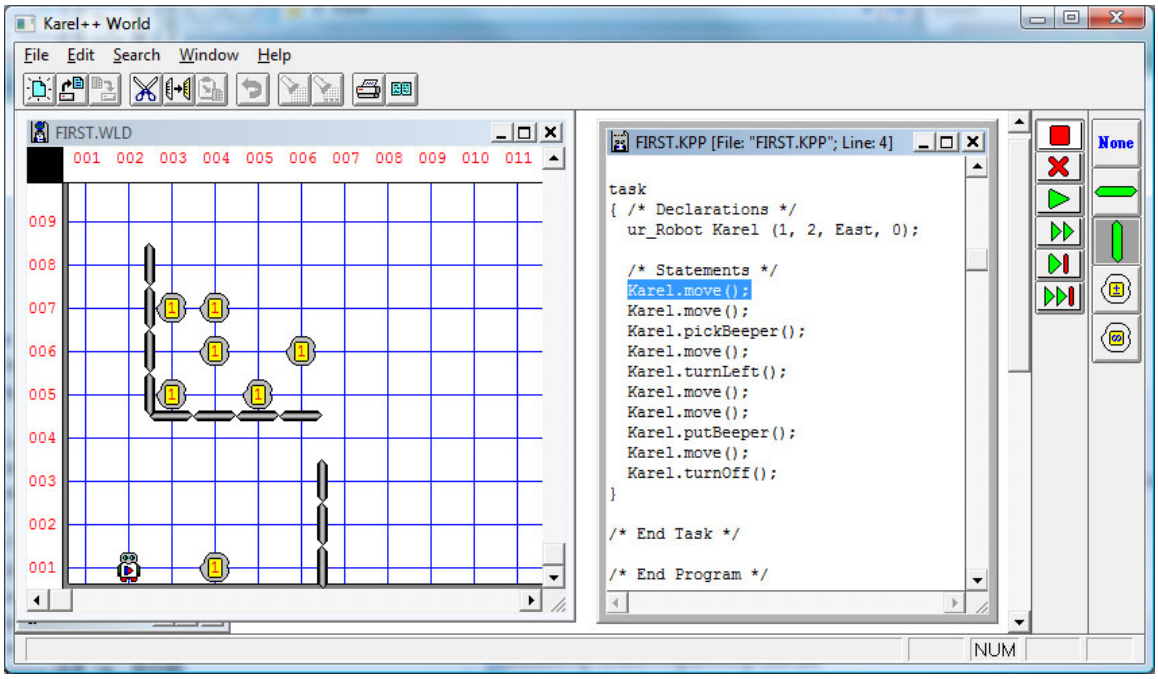

Figure 1. Karel++ programming environment. The world is displayed on the left with walls and beepers, and the robot near the lower left corner. The programming instructions are displayed on the right.

Karel++ Program

The Karel++ program was developed in the mid 1990's, for Microsoft Windows, and has been used successfully on 32 bit editions of Windows Vista. Karel++ did not come with a setup or install program. A user simply copied the program files to a folder and ran the program. Before Windows 98 second edition, directory (folder) and file names were limited to eight characters plus a three character extension in the form "xxxxxxxx.xxx". Spaces were not allowed in file names. Like programs of its era, Karel++ had this limitation. If a longer folder or file name was used, it would appear in Karel++ with the file name truncated at six characters and "~1" typically added to the file name. Students were required to have a USB jump drive for storing files and were instructed to use folder and file names with no more than eight characters and no spaces. The default file extension of a Karel++ world was ".wld". Since worlds were created and saved only in Karel++, long file names or names with spaces were not accepted and did not present any complications. Karel++ does not include a source code editor, so a text editor, such as Windows Notepad or the shareware TextPad was required. The normal file extension for source code files was ".kpp" which was a text file, although any extension can be used. The main source of confusion for students using Karel++ was the file name limitation. When using a text editor, if students used a longer file name or inserted a space, the file name would display differently in Karel++ than in other Windows programs. Otherwise, students found the Karel++ program easy to use.

When a source code file was loaded into Karel++, it was automatically checked for errors. If an error existed, Karel++ displayed an error message and the line number of the program where the error was detected. A text editor with a feature to display line numbers was very convenient when debugging programs. Students typically forgot syntax, such as a semicolon at the end of an instruction. They quickly learned the value of attention to details in programming and learned

that computers were not mind readers. Once a program was error-free, it could be executed at normal speed or in step mode. In step mode, Karel++ allowed a user to step through a program line by line. In this mode, a line of the source code was highlighted and when the user clicked the step button, the line was executed and the next line of code was highlighted (Figure 1 depicts the step mode with the "Karel.move()" instruction highlighted). This ability to step line by line through a program was very important to help students find their logic errors.

Karel++ Text

The text, *Karel++ A Gentle Introduction to the Art of Object-Oriented Programming* (Bergin, Stehlik, Roberts, and Pattis, 1997), was divided into six chapters, with the first five being used for our course. Chapter 1 described the robot world, robot capabilities, and the terms task and situation. Chapter 2 described the five primitive instructions (tunLeft, move, pickBeeper, putBeeper, and turnoff), provided detailed examples of programs, described program errors, and included tips for debugging. Chapter 3 detailed the procedure for creating a new class and new instructions, including inheritance and polymorphism. This chapter described the problem solving strategy of stepwise refinement (or hierarchical decomposition). Chapter 4 covered "if", "if/else", and nested "if" instructions, eight built-in Boolean instructions, procedures for creating Boolean instructions that return a true or false value, and using Boolean instructions within an "if" instruction. Chapter 5 covered the repeating instructions of "loop" and "while."

The text was used extensively throughout the Karel++ unit of the course. As each of the five chapters are assigned for reading, students were required to complete a homework assignment of 25 true/false, multiple choice, or matching questions related to the chapter. The text had examples that students could use as a guide in completing the lab programming assignments.

Karel++ Programming Unit

The required "Introduction to Control Programming" course has been taught during the spring term for six consecutive years. The course was divided into two units. The first unit covered Karel++. The objectives for the Karel++ unit were two-fold.

- Students would be able to write simple computer programs and have an understanding of concepts for basic interaction with programming professionals
- Students would improve their problem analysis and problem solving skills

Students were not expected to be computer programmers, but they could realistically develop simple control and logic programs, discuss needs and strategies with a computer programmer, and improve their problem solving skills. Students with further interest in programming were directed to courses offered in other colleges within our university.

The Karel++ unit consisted of eleven lectures, ten programming assignments (labs), two exam review days, and two exams (see Table 1). The first exam covered the first three chapters of the text and the second exam covered the next two chapters. The first labs were relatively simple to provide quick success for students. Each subsequent lab generally increased in complexity.

Table 1. Listing of course content as presented each day of class.

Class Day	Course Content
1	Course Introduction Assignment: Homework for Karel Chapter 1 Assignment: Homework for Karel Chapter 2
2	Lecture: Karel Introduction Assignment: Homework for Karel Chapter 3 Lab 1 : Karel 1st Program
3	Lecture: Karel Programs Lab 2: Simple Programs
4	Lecture: Extending Karel's Language Lab 3: Define New Robot Lab 4: New Robot Class Demonstration: Using TextPad to write new class for robot
5	Lecture: Number Systems Lecture: Software Design Lab 5: Harvester Robot
6	Lecture: Object Oriented Lecture: Multiple Robots Lab 6: SciFi Robots
7	Review for Exam 1
8	Exam 1 Assignment: Homework 4 - Karel Chapter 4
9	Lecture: Conditional Instructions Assignment: Homework 5 - Karel Chapter 5 Lab 7: Better Harvester
10	Lecture: Conditional Instructions (Cont'd) Lecture: Problem Solving Strategy Lab 8: Using Predicates
11	Lecture: If Variations Lab 9: Using Ifs
12	Lecture: Repeating Instructions Lab 10: Loops
13	Review for Exam 2
14	Exam 2

The class was scheduled to meet for 75 minute periods, two days per week. Each class period typically included a lecture slide presentation and time for students to work on programming (lab) assignments. On most days, the computer lab and instructor were available to the students an additional 35 minutes at the end of the class. The extra lab time benefited many students by helping them keep up with the lab assignments. Rather than have the students turn in labs for evaluation later, we found it more efficient to review and evaluate the programs with the students during the lab time.

The lectures included the use of Microsoft® PowerPoint presentations and code examples written on a white board and covered the following topics and related labs:

1. The Karel++ Introduction lecture introduced the Karel++ programming environment, robot capabilities, running an existing program, modifying a world, and modifying a program. The

related lab simply consisted of running an existing program, adding borders (walls) and plants (beepers) to a world, and making a simple modification to a program. This objective of this lab was to acquaint students with Karel++ and allow them to achieve immediate success in running a program.

2. The Karel Programs lecture explained general programming terminology, object oriented programming terms, description and example of classes, language syntax, reserved words, each instruction in the program for the first lab, and finding errors. The related lab consisted of writing a program for a robot to perform three simple tasks: walking a square fence line, pick up a newspaper (beeper) on the front porch, and ascend and descend a set of stairs. Besides allowing a student to develop proficiency in writing simple programs, the labs were intended to demonstrate the tediousness of writing programs with only the primitive instructions. At the end of the lab, students were asked to list instructions they wished were available for a robot. Having students think about new instructions that could be created from the primitive instructions prepared them for the next lecture and lab.

3. The Extending Karel's Language lecture stressed the importance of being able to extend a robot's language to reduce the number of repeated instructions and develop reusable code. The syntax of declaring a new class and defining new instructions was presented. Naming conventions for classes and instructions were presented with the textbook model of camel casing being recommended. Students were required to use descriptive names for full credit on subsequent labs. There were two labs related to this lecture. The first lab required the students to define six new instructions for a new class of robot. As an example, a turn right instruction was defined as consisting of three turn left instructions. Each new instruction was required to consist of primitive instructions or other new instructions.

   Once this lab was evaluated, students started on the second lab which requires writing code for the new class and instructions, and writing code to demonstrate the use of each new instruction. The instructor guided the students by writing code for a new class and the turn right instruction using the instructor computer and data projector.

4. The Number Systems lecture introduced the binary, octal, and hexadecimal number systems, explained why they are used in programming, and detailed how to convert between these systems and decimal. Once the students completed the conversions manually, the instructor demonstrated using the scientific mode of Windows Calculator to convert between the number systems. Binary is the number representation used within computers. Hexadecimal is a type of binary shorthand for 4 bits of data and is encountered when interfacing some microprocessor based equipment. Octal is not as common, but has been used to represent alphanumeric characters. Although students did not use these number systems with Karel++, we believed students in a programming class should be introduced to the number systems.

5. The Software Design lecture was one of the most important for the course. This lecture prepared students to begin writing more complex programs and introduced problem solving strategy. Programming guidelines of code being easy to read and understand, easy to debug, and easy to modify were presented. A good program was defined as a composition of easily understood parts, using a rule of thumb that each part contained five to ten instructions. The ability to ignore details that are temporarily or no longer relevant to a specific part was presented as a powerful aid to programming and debugging.

   Problem solving steps of defining the problem, planning the solution, implementing the plan and analyzing the solution are introduced and examples were presented. The stepwise

refinement method of developing a solution was presented using the example of harvesting a six by six grid of plants (see Figure 2). Students were encouraged to discuss potential solutions. Typical options were harvest a row, return to starting point, move to next row, and repeat; harvest going in both directions; and harvest the perimeter, moving in one row and repeat. The more common solution was to harvest going one direction, move up a row and harvest in the opposite direction until the harvest was complete.

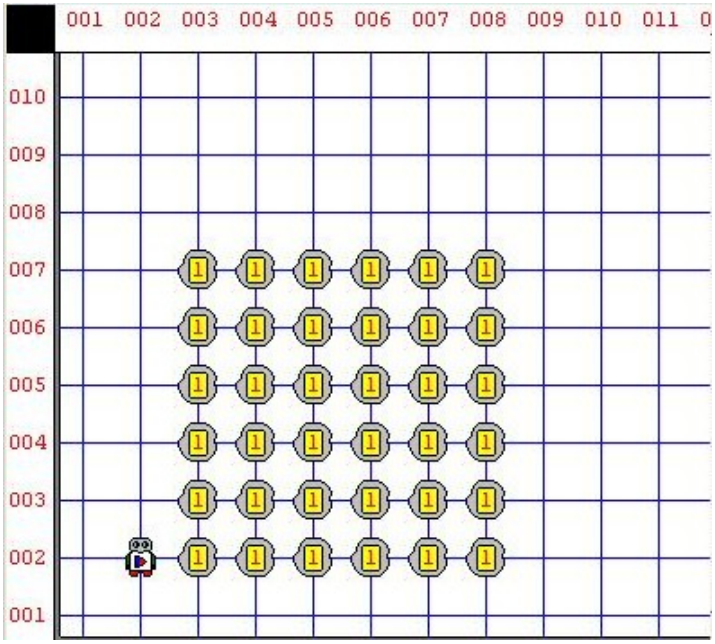

Figure 2. Initial situation for a robot program to harvest a six by six grid of plants.

Using stepwise refinement, students were encouraged to first look at the big picture. The task was to harvest the field, so a new instruction called harvestField could be written. The harvestField instruction could consist of thee harvestRound instructions. The harvestRound instruction could consist of harvestRow, endTurnLeft, harvestRow, and endTurnRight. The harvestRow consists of a series of move and pickBeeper instructions. The endTurnLeft and endTurnRight instruction used move and turnLeft or turnRight to position the robot for the next harvestRow. By using this method, students were encouraged to first look at the overall problem and give it a name, then divide it into sub problems and give them each a name, and continue this process for each new sub problem until a solution was derived. Students were encouraged to analyze their solution to determine its strengths and weaknesses and revise as needed. This stepwise refinement method is suitable for solving a wide range of problems beyond programming. We found it important to involve students in discussing stepwise refinement solutions as new labs were introduced to help them assimilate this new problem solving tool. The lab related to this lecture is a harvesting lab similar to Figure 2.

6. The Object Oriented lecture provided more details and examples on the object-oriented programming concepts of inheritance and polymorphism. Inheritance was described using pickup trucks, in that if our favorite manufacturer said they were designing a top secret pickup truck, we would still know a lot about it. There are certain features of a pickup truck that have been inherited for generations of pickup trucks, such as a cab with at least two doors and a cargo bed. Inheritance was stressed as a time saver, as a new class can inherit all the instructions of a previously written class. Polymorphism is the ability to have the same instruction name in different classes that is interpreted differently in each class.

Polymorphism was not required for any of the labs, although some students used it unintentionally.

7. The Multiple Robots lecture introduced the use of more than one robot to solve a problem. The related lab had the students program multiple robots to mow the word “HELLO” into a soybean field (see Figure 3). Robots could be specialized, with each one designed to mow a different letter. This problem required extended use of stepwise refinement to break the problem down into reusable parts.

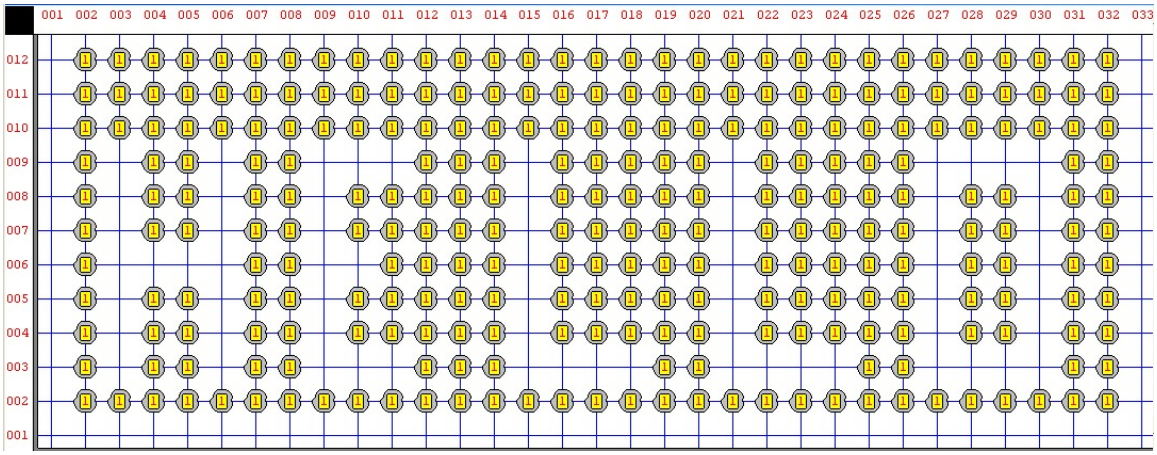

Figure 3. Completed “SciFi” lab using multiple robots to mow the word “HELLO” in a soybean field.

8. The Conditional Instructions lab introduced students to the “if” instruction and the primitive Boolean instructions to write new Boolean instructions (see Figure 4). Examples of “if” instructions were presented and students voiced their opinion whether the resulting condition was true or false. Karel++ has a negation operation (“!”) which can be used in writing or using Boolean instructions. The introduction of Boolean instructions and use of them with “if” tended to be confusing for about half of the students, particularly when negation was introduced. Patience was needed at this point to reinforce the content of this lecture with several examples. We found it important to review this lecture at the beginning of the next class meeting.

```
Boolean Robot_Class :: rightIsClear()
{
      turnRight();
      if ( frontIsClear() )
      {
            turnLeft();
            return true;
      }
      turnLeft();
      return false;
}
```

Figure 4. Karel++ code for a Boolean instruction to return true or false depending on whether the right was clear.

The related lab was similar to the harvesting problem, except that plants were missing on some coordinates. If a robot attempts to pick up a plant (beeper) where none exist, it will cause the robot to turn off (terminate the program). Students added the “if” and a Boolean instruction to their prior harvest program to solve this problem.

9. The Problem Solving Strategy lecture introduced a general process flow of input, process, and output. Many problems were easier to solve, if a student visualized the output and then worked backward to which inputs are required and what process is needed to arrive at the desired output. Often students would begin writing code without clearly visualizing the end result or output. This lecture was important to encourage students to visualize the output before writing code. Example problems of calculating average exam score, harvesting a field, payroll check writing, grain farming, and breakfast cereal production were used in a class discussion. The strategy of visualizing the output first was applied to writing Boolean instructions for use in Karel++ programs.

   The related lab required students to program three new instructions. They had to determine if each instruction was a Boolean type or just used a Boolean instruction. The students also wrote code to demonstrate the proper operation of each of the three instructions.

10. The If Variations lecture introduced “if/else”, nested “if” instructions, and complex “if” tests. Karel++ does not support “and” or “or” operators. Students learned to use nested “if” instructions for “and” conditions. An example problem was replanting and thinning of plants in a garden. If a coordinate was void of plants, one was planted. If a coordinate had more than one plant, the additional plant(s) was removed. Examples of interpreting and simplifying conditional instructions were discussed. The related lab was a dead-end, avenue-paving, robot problem. The robot must move along a street and enter each avenue. If the avenue was a dead-end, the robot paved the avenue, otherwise it moved to the next avenue (see Figure 5). This lab also required students to hand write code for “and” and “or” conditional tests.

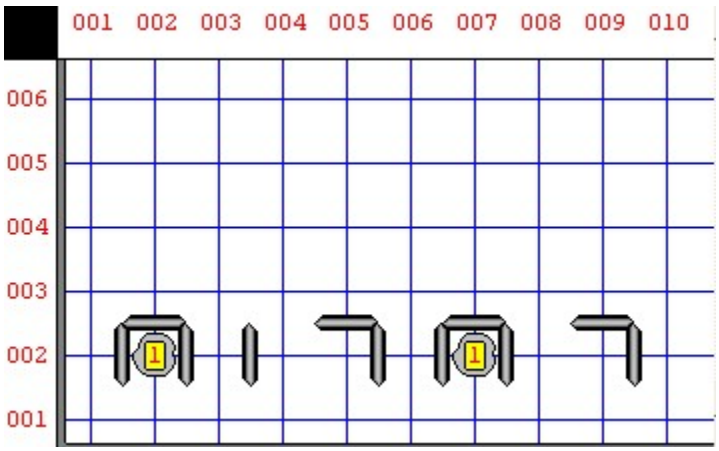

Figure 5. Completed dead-end street paving lab. The robot started at coordinate 001,001 and moved to the right along street 001. It entered each avenue and paved (placed beeper) the ones that were dead-ends.

11. The Repeating Instructions lab introduced the “loop” and “while” instructions. “Loop” was the only instruction in Karel++ that used a numeric parameter—for the number of times to execute a loop. The “while” instruction was introduced as the most powerful, since it combined the conditional features of “if”, with the repeating feature

of “loop.” Detailed examples on building and testing a “while” instruction were presented. Students were presented with several code snippets using “if” and “while” to interpret. The related lab had two parts. The first problem was to plant a square perimeter of shrubs. This problem could be readily solved using a loop with a nested loop. The second problem was described as a steeple chase with varying heights of obstacles positioned at different intervals (see Figure 6). The solution to this lab used “while” with a nested “if” with a nested “while.” This final lab included use of new language features, insightful problem analysis, and stepwise refinement to reach a solution.

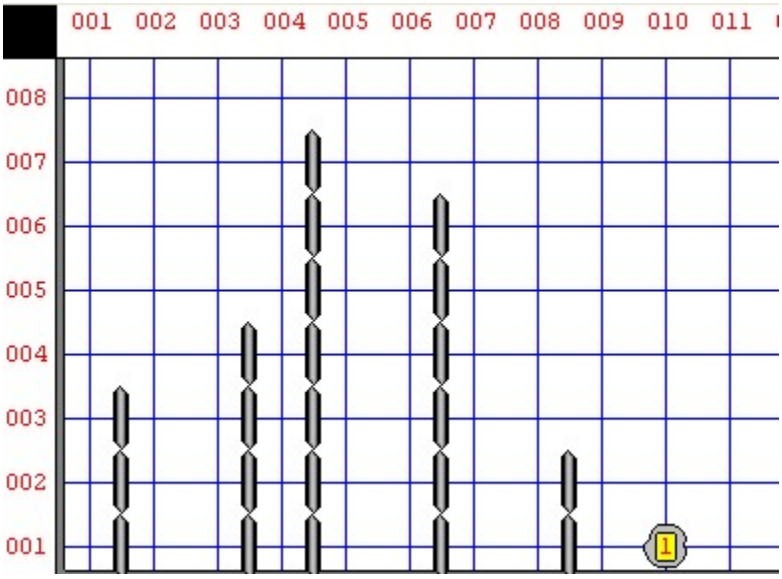

Figure 6. World for steeple-chase lab. Robot must move from 001,001 along street 001 and climb over any hurdles until reaching the finish line (beeper).

Student Feedback

Two evaluation instruments were used to gain student feedback. The first instrument was our university’s standard instructor and course evaluation (ICE) form. Since the Karel++ unit is only one half of the course, the ICE results are not exclusive to Karel++. We did assume that if students had major concerns about the Karel++ unit, those concerns would be reflected in the overall course evaluation. Over the past five years of teaching the course, the instructor ratings averaged 4.5 on a 1 to 5 scale, with 5 being excellent. The average overall course rating was 4.2 on the same scale. Course enrollment averaged 17 per term over the same time period. These ratings were considered above average in our department. We have been pleased with the ICE results, particularly considering this was a required course.

Students could also indicate their level of agreement (strongly agree, agree, neutral, disagree, and strongly disagree) with specific statements about the course. Averaging the past five years of responses, students agreed that the “course was a good learning experience”, the “course was very interesting”, and the “required work was appropriate.” Student responses averaged between neutral and disagree for the statements “course material was too difficult” and “I was often confused.” We were satisfied with these results. If all students strongly disagreed that the

course was too difficult or that they were confused, we probably wouldn't be challenging the better students.

The second evaluation instrument allowed students to write their own responses to questions, rather than choose a response. This instrument was used at the end of the most recent term. When asked "what part of the course challenged you the most," 84% of students indicated the Karel++ labs or at least the last Karel++ labs challenged them the most. When asked "what part of the course challenged you the least," 28% of students wrote the beginning Karel++ labs. When asked "do you think the course helped improve your problem solving skills for more complex problems?" 78% said yes and 22% said no. When asked "in what ways, if any, did the course cause you to analyze problems differently?" 80% of students indicated a difference, with responses such as:

- learned many different ways to start at the problem and work backwards
- use step by step methods
- analyzing entire problem before working on solution
- look at the smallest details and take my time
- check my work over more carefully

Discussion and Recommendations

After six years of integrating an introduction to computer programming into a new "Introduction to Control Programming" course, we were pleased with the result. The use of Karel++ to teach introductory programming helped us achieve objectives of students learning to program, and improving their problem analysis and problem solving skills.

The Karel++ programming environment was essential to the success of this required course. The robot paradigm allowed us to meet Biermann's first rule of having interesting applications for teaching programming to nonmajors (Biermann, 1994). The robot applications revealed the lack of computer intelligence, as students learned they were responsible for controlling every detail of the robot's behavior (Biermann, 1994). We expect students successfully completing the Karel++ unit to be less likely to view an automation or decision making system as a "black box". Their exposure to programming provided them with an understanding of potential programming within an automation or decision support system.

We followed the recommendations of Gries (1974) and Biermann (1994) and included problem solving methods in our course. The experience of dividing a complex problem into smaller, easier to understand parts is possibly the best outcome of the course. Nearly 80% of students said the course helped improve their problem solving skills for more complex problems.

Other descendants of Karel have been developed in recent years, including Jeroo (Sanders and Dorn, 2003), ActionScript (Crawford and Boese, 2006), and objectKarel (Xinogalos, Satratzemi, and Dagdilelis, 2006). Enhancements in objectKarel include an integrated source code editor, class declaration and constructor forms, display of state and properties, and extended tutoring information. So far, we continue to use Karel++, but do like the integrated source code editor feature of objectKarel.

We recommend Karel or one of its descendants for instructors looking for a method of introducing computer programming, with an emphasis on problem solving. Instructors with experience in any programming language should be able to learn and teach Karel++.

References

Becker, B.W. (2001). Teaching CS1 with Karel the Robot in Java. *ACM SIGCSE Bulletin, 33* (1), 50-54.

Berge, O., R.E. Borge, A. Fjuk, J. Kaasboll, and T. Samuelsen (2003). Learning Object-Oriented Programming. *Norsk Informatikkonferanse NIK'2003 , Tapir Akademisk Forlag*, 37-47.

Bergin, J., M. Stehlik, J. Roberts, and R. Pattis (1997). *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming.* New York: John Wiley & Sons.

Bergin, J., M. Stehlik, J. Roberts, and R. Pattis (2005). *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java.* Dream Songs Press.

Biermann, A.W. (1994). Computer Science for the Many. *Computer, 27* (2), 62-73.

Buck, D., and D.J. Stucki (2001). JKarelRobot: A case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum. Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, (pp. 16-20). Charlotte, NC.

Cooper, S., W. Dann, and R. Pausch (2000). Alice: A 3-D Tool for Introductory Programming Concepts. *Journal of Computing Sciences in Colleges, 15* (5), 107-116.

Crawford, S., and E. Boese (2006). Actionscript: A Gentle Introduction to Programming. *Journal of Computing Sciences in Colleges, 21* (3), 156-168.

Gries, D. (1974). What Should We Teach in an Introductory Programming Course? *ACM SIGSCE Bulletin, 6* (1), 81-89.

Harper, J., P. Buriak, and B. Hitchings (2001). University Faculty Perceptions of the Technology Education Needs in Food, Environment and Natural Resources System. *ASAE Meeting Paper 01-8029.* St. Joseph, MI: ASAE.

Kelleher, C., and R. Pausch (2005). A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computer Surveys, 37* (2), 83-137.

Pane, J.F., and B.A. Myers (1996). *Usability Issues in the Design of Novice Programming Systems.* Retrieved Jan 25, 2006, from http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html.

Pattis, R.E. (1995). *A Gentle Introduction to the Art of Programming (2nd Ed.).* New York: John Wiley & Sons.

Sanders, D., and B. Dorn (2003). Classroom Experience with Jeroo. *Journal of Computing Sciences in Colleges, 18* (4), 308-316.

Xinogalos, S., M. Satratzemi, and V. Dagdilelis (2006). An introduction to object-oriented programming with a didactic microworld: objectKarel. *Computers & Education, 47*, 148-171.