Improving Student Preparation and Retention
With a Mid-Semester Supplemental Course Option

Dave Williams and Dwight Egbert
Western Nevada Community College / University of Nevada, Reno

Abstract

Otherwise capable students who lack sufficient preparation may lose interest, drop courses, and possibly withdraw from a technical program when confronted by the daunting task of learning the principles of basic algorithm development and programming technique in C++ or Java at the same time. Several weeks into an introductory programming course, a cohort of struggling students likely to withdraw can be readily identified. An alternate mid-semester course option with an emphasis on introductory algorithms has been developed in an attempt to retain and benefit these students.

The course is a dedicated problem-solving forum built around the less hostile and graphic-enabled Matlab programming language which focuses on the translation of useful and interesting problems into code. Potential CS drop candidates are strongly encouraged to add this course as they withdraw from the programming class with no additional fees (via the “drop/add” process). A more relaxed pace and the enrichment nature of the course provides a fresh start for struggling students, and the skills and confidence obtained in this course will increase the likelihood of their success in a conventional programming class in a future semester. Equally significant, the course maintains student participation in the technical curriculum and will therefore be likely to improve student retention. Other students seeking Matlab instruction, additional problem-solving development, or an introduction to elementary game programming are also invited to enroll. The course has also been accepted as a technical elective for non-engineering majors but is not applicable toward the college’s Engineering Technology or Engineering Science degrees.

The structure, curriculum, and class project used in the initial offering of the course are presented in this paper. The real centerpiece of the course was a collaborative class project: an implementation of the game of tic-tac-toe that can never lose a game to a human player. Every student was required to contribute a portion of the code that would function properly within the overall program. To assist other faculty in the formulation of similar projects, this semester’s programming project is described in great detail to present the many facets deemed critical to meet the objectives of the course. Also included are some suggested course modifications that will be implemented in future sections. Much of the material used in the course is available upon request for faculty who would like to implement a similar course at their institution.

Index Terms

Computer Programming, Matlab, Student Retention, Problem Solving, Tic-Tac-Toe
Introduction and Background

Introductory engineering technology students bring a wide variety of learning skills and technical preparation to their first courses. Those interested in a technical education are almost always drawn to computing hardware and programming languages well before they begin their post-secondary education. However, the degree of student preparation varies widely. Some students are accomplished programmers while others have little or no applied problem-solving experience. Most are comfortable with basic computer operation, but a subset of every entering class is uncomfortable with even these vital skills. Their mathematical preparation ranges from mastery of intermediate calculus concepts to required enrollment in remedial algebra courses. Similarly, most entering students are typically encouraged to enroll in the introductory programming class as early in their degree program as possible, as these courses serve as prerequisites for many that follow. Accordingly, this severely heterogeneous student mixture lacks a common foundation of basic mathematical and engineering material from which to create suitable introductory programming assignments. The situation is exacerbated by varying student objectives; while some will be pursuing programming-intensive degrees (including Computer Engineering Technology), others will not require further programming expertise and view their only required computer science course as something to be endured because it can’t be avoided.

The typical CS1 course in engineering technology programs introduces basic algorithm development using C++, Java, or a similar high-level programming language. The volume of new material typically covered in these courses is several times that of many other courses, especially in non-technical subjects. Students are faced with the simultaneous tasks of learning an unforgiving computer language (more daunting than most foreign languages) and the basic techniques of problem-solving at the same time. Those with a developed interest or aptitude in the subject usually excel while others are much more likely to struggle in one or both facets of the curriculum. Lack of student effort is to blame in some cases, but many others are simply attributable to the volume and complexity of the material to be covered in a single semester coupled with a lack of adequate pre-college preparation.

Every student must begin somewhere, and most correctly believe that the appropriate point of entry is with the introductory programming class. However, when the material in this course proves too much for some students to digest at once, many are rightfully discouraged and confused. Lacking another option, students withdraw from the course in lieu of failing and are reluctant to enroll in the same course next semester because they recognize that they will be no better prepared the second time than they were initially. If the course is required for an ET degree (as it is in most cases), it can effectively serve as a barrier to successful completion of their chosen program of study in the same way a moat encircles a castle. A teaching institution which does not provide a reasonable transition to the material in its introductory courses risks alienating a sizeable portion of its potential enrollment. When students can’t climb the castle walls, a drawbridge across the moat should be offered to provide the access they require.

Struggling students will endure an additional setback. By the time many realize the gravity of their situation, the semester deadline for adding new courses has passed (this date is usually within the first two weeks, which translates into approximately four lectures). Fees paid for the dropped course are forfeited and few, if any, other courses are available at that late date for a simultaneous drop/add action.
Summary of Desired Project Objectives

In light of these circumstances, this curriculum development project was undertaken with the following defined objectives:

1) To create an enjoyable but stimulating course in introductory programming concepts, problem solving, and algorithm development using a language stylistically similar to popular high level languages and which encourages modular programming style but with a less severe learning curve.

2) To structure the course so that it is both suitable for students not yet prepared for the typical CS1 curriculum but yet interesting and challenging enough for other technical students seeking additional programming experience and non-technical students enrolled in the class as a technical elective. Programming assignments and projects should focus on the solution of problems that do not require significant math or engineering background and, to encourage participation and maintain interest, would preferably involve something already familiar to students.

3) To engage in class exercises that build skills and confidence to better prepare them for successful completion of CS1 during the following semester, allowing their continued participation in computer science instruction and improving the possibility of retaining them as ET students. Whenever possible, group activities which build teamwork skills should be encouraged.

4) To schedule the course in such a way that students ready to drop CS1 may move seamlessly into the course after dropping with minimal loss of credit and fees and without significant disruption to their class schedule.

A First Incarnation

The prototype offering of this course was offered during fall semester 2003 at Western Nevada Community College in Carson City, NV. An existing college catalog entry for an introductory programming course that had not been offered in many years was suitably modified for the new offering:

CS 103: Computer Science for Engineers and Scientists (2 credits. Prerequisite: MATH 128 or consent of instructor). Introduction to programming in Matlab and applications software using individual computers. Elementary numerical methods and symbolic methods to solve problems in engineering and science.

Instruction was scheduled to begin in the sixth week of a fifteen week semester, leaving ten weeks of instruction for the course. Lectures were combined with some in-class lab work and were scheduled for two 75 minutes sessions per week (equivalent to three contact hours per week). Given the college’s requirement of fifteen semester contact hours per unit of semester credit, this class met all academic requirements for two semester credits.
The late start date was necessary to accommodate those students who attempted CS1 that semester but decided to drop when they realized that they are not yet ready for that course (accommodating the continued instructional needs of these students is one of the fundamental objectives of this course). With one additional 75 minute instructional period per week, the ten-week course could also be offered for three credits if so desired. For the initial implementation, two class meetings per week over ten weeks for two semester credits was selected.

Because the enrollment would not be known until instruction was about to begin, the days and times of course meetings were initially scheduled and publicized in the semester schedule as TBA ("to be announced"). The course was promoted to all students in the college’s Engineering Technology and Engineering Science programs during the first weeks of the semester, especially within the CS1 class. When the enrollment was finalized at the start of instruction, a meeting was held of all students enrolled and a mutually agreeable class schedule was devised. The small enrollment in the pioneer offering of the course made this a viable plan of action. However, with larger enrollments, establishing a set schedule in advance may be preferable.

Western’s engineering curriculum is fully articulated to the College of Engineering at the University of Nevada, Reno (UNR). Most notably, Western’s Computer Science classes have adopted many of the instructional modules produced by the Combined Research-Curriculum Development program in Computer Vision at UNR. These C++ instructional tools permit introductory and intermediate programming students (CS1 and CS2) to gain significant programming experience in tasks such as image processing and have met with great success at the college and elsewhere. Similarly, other courses within UNR’s College of Engineering have standardized on the use of Matlab, the commercial matrix-based programming language and visualization suite from MathWorks. Matlab’s programming language is quite similar to C and C++, allowing modular program construction using “m-files” with only minor syntactical difference at the level of this course. It also offers easy implementation of graphically-based input and output. Matlab was selected as the language to be used in CS 103 because of these strengths and its availability in the college’s Linux-based computer laboratory. However, other similar code suites, including at least one open-source project with some degree of Matlab compatibility, may also prove suitable for this course.

Lectures were held in the engineering computer lab with each student seated at a separate workstation. This allowed the instructor to lead the class in synchronized interactive exercises during lecture when helpful to the flow of instruction. All code provided by the instructor and later developed by students was placed in a publicly-accessible directory NFS-mounted on each workstation so that each student could copy the code to his or her home directory for testing and modification.

Basics of the Curriculum

While there are a number of good reference books on introductory programming in general and Matlab in particular, the decision was made not to select one as an official course textbook. Students were certainly permitted to purchase one for their own use if desired. In lieu of a text, students were provided with handouts specifically tailored to the material covered in the course when appropriate. Matlab has very comprehensive embedded help and reference material, and
students were repeatedly encouraged to use this utility when needed to build familiarity with on-line help systems and to wean them from the ingrained reflex action of turning pages in books.

As in the instructor’s other programming courses, the pedagogy of “demonstrate and expand” was utilized. This method begins with a code example provided by the instructor which accomplishes some basic operation. This code is dissected line-by-line for the class until its operation is fully understood. Students are then encouraged to run the code as supplied and compare its behavior to that expected, noting any problems or anomalies in its performance. An enhancement or expansion to the code is then proposed by the instructor, and students are tasked with modifying the code to provide that additional functionality. As often as possible, students are encouraged to work in small groups of two to three students each; the collaboration of several students provides an atmosphere in which even the most reserved students may build confidence.

The course began with an introduction to basic shell operations under the Linux operating system, as Linux was a new experience for all of the students enrolled in the course. Students were then given a short tutorial on Matlab\(^9\) as a hand-out and instructed to work through it individually. The tutorial covers basic language functions including data entry, numerical operations, elementary graphing functions, and creating and saving program files. Students were to repeat the short and simple examples and exercises in the tutorial working on their own for a short period to gain confidence typing Matlab commands and checking the results against those presented in the tutorial. Students with developed programming skills were given an additional exercise or two to occupy them until the rest of the class has completed the tutorial.

The first few weeks of the course focused on a series of straightforward algorithms designed to illustrate simple programming techniques. Armed with a preliminary exposure to basic Matlab operations, students were presented with a rudimentary m-file program which draws a straight line expressed of the form \(y = mx + b\) given the slope \(m\), the y-intercept \(b\), and the minimum and maximum values of \(x\) as input.

```matlab
close;                          % closes an open graphic window
m = input ('Slope of line: ');  % supply input data to program
b = input ('y intercept: ');    %
xmin = input ('Minimum x value for plot: '); %
xmax = input ('Maximum x value for plot: '); %
x = xmin:.1:xmax;               % create the x vector from the data
y = m * x + b;                  % create the y vector based on x
plot(x,y)                       % plot each point
grid on;                        % draw Cartesian grid for the plot
```

Figure 1. Simple Matlab m-file to draw a straight line given the slope and intercept
The code fills a pair of vectors with the abscissa (x) and calculated ordinate (y) values and plots the ordered pairs to a self-dimensioning graphic window. Students are given a few minutes in class to run the program and are then asked to comment on its functionality. Astute students will have discovered that this program is not well suited to draw a vertical line due to the requirement of supplying an infinite slope in numerical form. Accordingly, they are led to the conclusion that it is far from useful for most graphic applications.

The next code example is a function which provides something more, but not yet fully, useful:

```
function  plotline(x1,y1,x2,y2)
% plotline(x1,y1,x2,y2) connects (x1,y1) and (x2,y2) with a straight line.
m = (y2 - y1)/(x2 - x1);        % calculate the slope
if (x2 > x1)                    % fix step parameter so we can
    step = .01;                  % move either way on the x axis
else
    step = -.01;
end
xp= (x1:step:x2);               % step through x and fill x vector
yp = y1 + ((xp-x1) * m);        % calculate and fill y vector
plot(xp,yp,'-r')                % draw the line
end
```

Figure 2. Matlab function `plotline()` to draw a line segment given the endpoint coordinates

The methodology is basically the same, as the function still calculates and plots the equation of a straight line. This time, subsequent points are calculated based on the previous point coordinates and the slope of the line for a fixed domain increment. The utility of the function is greatly improved over the first program since the user may now specify the endpoints of the segment, making it preferable for drawing figures and, seemingly, even vertical lines. This example also introduces the utility of conditional control (`if-else`) as well as the proper format for m-files that will be called as functions by other m-file programs, including the use of arguments. The flexibility of Matlab allows this function to be invoked directly from the command window without a calling program by simply entering the arguments directly. For example,

```
plotline (2,1,4,7)
```

will draw a line segment connecting the points (2,1) and (4,7) in a new figure window.

Students are once again asked to test this code and comment on its performance. Hopefully, they will consider and attempt the problem task identified earlier (drawing a vertical line). Those who do will discover that even the improved version of the program will fail in this regard, but this time, for a different reason. Here, the calculation of the slope will generate a divide-by-zero error.
when \( x_2 = x_1 \). Clearly, a further improvement is still indicated, and the problem is given to students as a take-home assignment. Successful student solutions, along with the instructor’s version, are presented and discussed in a subsequent class. At this point, the students do not realize that this function will be an important component of their group project.

Defining the Class Project

Following several additional small exercises of a similar nature, but with escalating complexity, a semester project was presented to the class. Again, an important objective of the course is that all programming assignments be as non-technical as possible in order to maintain the focus on the development of problem-solving skills and algorithm development. After due consideration, it was decided that the entire class would collaborate on the creation of a tic-tac-toe program that would never lose when played against a human opponent.

The game of tic-tac-toe is exceedingly simple, yet the conceptualization of a computer program to play a proper game can at first seem overwhelmingly complex. The class was told that when played by equally skilled and attentive players, the game will result in a draw each and every time. The only way a player can lose a game of tic-tac-toe is by making a poor move. One student politely disagreed with this assessment of the game and claimed that he had seen games of tic-tac-toe that could not have been successfully defended by the losing player. That is, he was sure that there was a way for one player to force a victory over the other regardless of all efforts to prevent that from happening. The contention that the project objective (that the computer never lose) could not be achieved provided an interesting undercurrent to the project. The instructor sensed that other students, while mostly convinced that the instructor’s assessment was probably correct, may have harbored some slight reservation (or secret hope) that the dissenting student may be right. This piqued their interest in the project and is believed to have resulted in greater participation than if the problem had been less intriguing. They approached it as somewhat of a friendly challenge rather than as an exercise devoid of interest. If the instructor was satisfied that their functions were correct, students would only need to demonstrate one case where the computer could be beaten to disprove the instructor’s assessment of the problem. On the other hand, the onus was placed on students to insure that any game lost by the program was not attributable to their poor programming. In short, the instructor insisted that the computer simply not be allowed to make a bad move and encouraged the class to challenge the premise of his algorithm within that context.

With 9 squares on the board, there are a total of 362,880 (9!) possible game scenarios (nine distinct first moves, followed by eight second moves, seven third moves, etc.). Facing this huge number of possibilities, students are initially shocked at the enormity of the problem and doubt that they will be able to complete a significant portion of the assignment in their lifetimes. This, of course, is an ideal way to begin the illustration of the power of elegant algorithm design. Once students realize that the number of distinct moves is but a tiny fraction of the theoretical maximum number (including the fact that there are really only three unique initial moves), they begin to anticipate the power of a well-programmed solution. They will soon learn a full complement of algorithm refinements that will reduce the problem to a shadow of its former apparent complexity.
Crafting the Methodology

Two full class sessions were required to guide the class to an overview of the most appropriate algorithm for this project. Many tic-tac-toe boards were hand-drawn on the lab’s whiteboard to illustrate the important concepts involved. However, in the end, the students were able to approach the project with a complete understanding of how it should be implemented.

It was decided that the game would always allow the human player to move first, since many people erroneously perceive an advantage in doing so. For a game that won’t lose, it was deemed far more genial to afford the human player every possible advantage, whether real or imagined. The player always denotes his/her moves with an X, while the computer would always play O.

The first task was to establish a topology for the game board; Figure 3 presents the simple board layout that was used for the program. As introduced above, there are only three possible first moves: a corner square, a side-center square, and the board center square. Six of the nine first-move possibilities can be made equivalent to other moves by a rotation of the game board. For example, a first move to square 3 followed by a 90° counterclockwise board rotation is identical to a first move in square 1. The two remaining corners only require different rotations to be equivalent to an initial move in square 1. Similarly, a first move in square 4 with a 90° clockwise board rotation is identical to a first move in square 2, and rotations identical to those used for the corner squares create equivalence for moves in squares 6 and 8. A first move in the center square is unique, so no rotation applies for the case where the initial move is made there.

![Figure 3. Tic-tac-toe board topology](image)

Accordingly, it was proposed that the control algorithm be limited to handle initial moves only in squares 1, 2, and 5. Other first moves would be evaluated by the program and the appropriate rotation applied to convert those first moves, and all that followed in that game, to the equivalent positions as if the first move was one of the three allowed. Likewise, each computer move made while considering the game according to its view of the rotated board would be converted to the player’s version of the board before the move is displayed. While the human player would always see the board correctly based on his/her first move, the computer would always be
dealing with the board as though the first move was 1, 2, or 5. In this manner, the game will accommodate any first move and display it correctly for the human player but will translate that move into one of only three cases that will be programmed. This simplification eliminates 241,920 possible game scenarios (two-thirds of its initial value of 362,880), which represents a very substantial improvement. The rotational translations are depicted in Figure 4.

```

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

0° rotation

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

90° clockwise rotation

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

180° rotation

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

90° counterclockwise rotation
```

Figure 4. Rotation schemes to translate board for reduced-scenario computer algorithm

Next, students were confronted with the question of how the computer should evaluate the game status in order to choose its next move. Some students envisioned a complex algorithm that somehow evaluated the potential for a win or a loss based on an overall positional situation that would allow the program to base its move on the relative strength or weakness of each possibility. Given some time to ponder this approach and how they might be able to translate it
into code, they correctly concluded that it was far beyond their developing abilities. The class soon recognized that if the number of possible moves could be reduced sufficiently, the simplest alternative would be to uniquely program the computer’s best response to every human move, resulting in the consideration of “pairs” of moves. In other words, for each possible human move, there would always be a carefully pre-determined computer move in reply. The drawback to this approach would be that the computer’s moves could be readily predicted by the player, but this was judged to also be largely true in the case of two skilled observant humans playing each other. Therefore, it was agreed that this would not markedly diminish the user’s playing enjoyment. This approach also contributes to a great reduction in the number of possible moves to be considered. After the player’s first move and the computer’s reply move, there would be seven possible squares for the player’s second move. All eight second moves, each representing a whole “branch” of possible scenarios, were eliminated by hard-coding the computer’s single reply move into the program. When the user’s second move and the computer’s reply move have been made (eliminating six program branches in the process), only five empty squares would remain, followed by three, and then by one. The player’s final move was judged to be superfluous since the drawn game can be determined with 100% accuracy after 8 moves.

Requiring the user to make an obvious, unproductive last move in a drawn game simply to fill the board was evaluated and was found to be detrimental to game play. Accordingly, drawn games terminate with a notice of the draw and one empty square on the board. For each of the three initial moves, there will only be a maximum of 105 moves (7 * 5 * 3 ) until the board would be complete filled. The game will always end at or before the point that 3 + (3 * 105) = 318 total pairs of moves had been completed. Of course, the program must trap for illegal moves and not allow the player to select a square already used in a previous move.

While a far cry from the original 362,880 possible game scenarios, the algorithm’s simplification of the problem is not yet complete. Many of the scenarios remaining will result in the player’s loss before all the moves in that particular branch were completed. Under many scenarios where an inattentive human player makes a bad move (example below), the computer might win as early as its third move (the sixth move overall), leaving three unfilled squares on the board. In fact, after two moves by each side (four moves total), four of the five remaining moves available to the user (80% of the remaining possibilities) will result in his/her loss on the next move (see Figure 5). Each similar case reduces the number of scenarios remaining to be considered. Also, move combinations such as 1-5-2-3 and 2-5-1-3 (the squares for X’s and O’s alternating moves) will result in the same squares being filled with the identical marks, leaving the rest of the play for both scenarios identical. In the end, the final algorithm reduced the practical size of the game space to less than 0.08% of the original problem, simplifying the solution of this problem to something well within the programming ability of the class.

Demonstrating how the power of a well-designed algorithm reduces a seemingly-massive problem into a much simpler one is an invaluable lesson for the budding programmers. It provided a comprehensive look at many techniques that are used in the development of numerical algorithms for technical applications, all in the context of a simple computer game familiar to every student. In that sense, it is an excellent instructional vehicle for introductory programming students. Following the formulation of the process, the program was coded by teams of students working cooperatively as a part of a class-wide software development effort.
Coding the Program

The class was broken into three groups, and each group was assigned one branch of moves in the
tree descending from each of the three unique first moves (squares 1, 2, and 5). Due to the size of
the class, the short amount of time remaining in the semester, and the fact that this was the first
time through the project, the instructor contributed several sections of the code, including the
function which prompted for the user’s first move and called the top-level function of each
branch. This code illustrated the format to be followed in all subsequent functions to maintain
coding consistency.

The full presentation and description of the program is beyond the scope of this paper. However,
details of the algorithm and several small sections of code are provided to illustrate their relative
simplicity and the advantages of coding this problem in the modular style using a multitude of
easy-to-write short functions which are far less daunting to students than a longer, complete
program.

Students were required to use a separate function for each level (pair of moves) in their branch of
the tree. The names of these functions were descriptive of their place in the tree, prefaced by a
single letter corresponding to the name of the team owning that branch (since function names can
not begin with a number). For example, if the user selected square 1 to begin the game, the
computer’s reply move was square 5, and control passed from the primary input function to
function $k_{1,5}.m$, the upper-most function in the branch of the tree corresponding to an initial
move in square 1. If the user then selected square 2, the computer replied by moving to square 3
and program control passed to function $k_{1,5,2,3}.m$. This process continued until one of two
results were obtained: either the computer was able to win the game, or the game was drawn.

Before the student teams began writing their series of functions, they were encouraged to
thoroughly explore all of the possible game scenarios and to carefully select each reply move.
Every pair of moves would not only determine the future course of the game and would also
determine the names of the functions that were subsequently called. Therefore, an erroneous
computer move would not only compromise program operation but would have function-name
ramifications for the remainder of that tree. Correcting any such errors would necessitate re-
naming and re-writing much of what followed, so the importance of the programmer’s version of
“measure twice, cut once” was stressed.

Students acquired a thorough understanding that proper algorithm development requires full
command of the process to be undertaken. The tic-tac-toe program was implemented by student-
authored control functions which called a number of ancillary functions to perform all program
operations:

1) Prompt the user for input by reading the mouse cursor position when a left-click was
registered and determining the desired square for the user’s move ($\text{turn()}$);

2) Translate the user’s move to accomplish the rotation that converts the user’s view of
the board to the reduced-scenario model used by the computer, when necessary
($\text{in_t()}$);
3) Translate the computer’s move back to the user’s view of the board from the reduced-scenario model used by the computer, when necessary (out_t());

4) Draw the X and O marks in the designated squares on the board displayed to the user (draw_x() and draw_o());

5) Display game results (win()) and error messages (illegal()) to the user.

```matlab
move = turn;        % read input
move = in_t(move);  % rotate
if move == 4;
    draw_x(4);
    draw_o(7);
    win;            % game over
    return;
elseif move == 6;
    draw_x(6);
    draw_o(7);
    win;
    return;
elseif move == 7;
    draw_x(7);
    draw_o(4);
    k_1_5_2_3_7_4;   % call this
    return;
elseif move == 8;
    draw_x(8);
    draw_o(7);
    win;
    return;
elseif move == 9;
    draw_x(9);
    draw_o(7);
    win;
    return;
else
    illegal(mfilename);
end;
```

Figure 5. Sample student function k_1_5_2_3.m

Each of the student functions used a comprehensive if-elseif-else structure to test each user move possible at that point in the game and then invoke the appropriate reply. While similar in appearance, each of these functions contained the series of function calls, tests, and reply actions necessary to correctly continue the game to its conclusion. The function k_1_5_2_3.m is presented in Figure 5 as a representative sample.

The project also incorporates as much of the course’s introductory material as possible. In particular, the draw_x() function (Figure 6) uses the final version of plotline() from earlier class exercises to draw all four extensions of the X emanating from the center of the square, as does the function draw_grid() which creates the four-lined tic-tac-toe game board.

One of the real strengths of Matlab for this course is its integrated graphic capability. The visual nature of the game of tic-tac-toe requires an appropriate visual display. Although an “ASCII-art” depiction of the game board might suffice, Matlab’s ability to accept graphical input from its output figure windows allows the human player to interact with the game board in essentially the
same way he/she would with another human player. Pointing to the desired square and clicking anywhere within that square registers a move; the input is read and converted to the correct square number by the function \texttt{turn()}, which is depicted in Figure 7.

```plaintext
function draw_x(square);
% draw_x() - places an X on the board in the designated square

square = out_t(square);

if square == 1;
    h = -6; v = 6;
elseif square == 2;
    h = 0; v = 6;
elseif square == 3;
    h = 6; v = 6;
elseif square == 4;
    h = -6; v = 0;
elseif square == 5;
    h = 0; v = 0;
elseif square == 6;
    h = 6; v = 0;
elseif square == 7;
    h = -6; v = -6;
elseif square == 8;
    h = 0; v = -6;
elseif square == 9;
    h = 6; v = -6;
end;

plotline(h,v,(h+2),(v+2));
plotline((h-2),(v+2),h,v);
plotline(h,v,(h+2),(v-2));
plotline((h-2),(v-2),h,v);
```

Figure 6. Function \texttt{draw_x()} calling \texttt{plotline()} from earlier class exercises

Pulling It All Together

As the students’ work progressed, their functions were added to the central repository containing all the functions being simultaneously developed. A shell script was provided which would update all of the files in each students’ home directory so that they would all have access to the entire code base as it was developing. The teams were encouraged to work on small segments of their particular branch until they were satisfied with the results, only then proceeding to the next segment. Slowly but surely, the pieces were assembled until the code suite was complete.

As expected, there were problems. Some moves that should have been considered were not, and there were a few poor computer reply moves included in several student functions. Learning how to resolve these problems is, of course, just as vital an exercise for programming students as is learning how write new code. One of the more interesting activities for the instructor was to
function square = turn();
% turn() - reads mouse position and returns appropriate square

h = text(-7.2,-10,'Your move . . . click on any empty square',... 
       'BackgroundColor',[.7 .9 .7], 'EdgeColor','black', 'EraseMode',
       'background');

[x,y] = ginput(1);
delete(h);

if (x < -3) & (y > 3);
    square = 1;
elseif ((x > -3) & (x < 3) & (y > 3));
    square = 2;
elseif (x > 3) & (y > 3);
    square = 3;
elseif (x < -3) & (y > -3) & (y < 3);
    square = 4;
elseif (x > -3) & (x < 3) & (y > -3) & (y < 3);
    square = 5;
elseif (x > 3) & (y > -3) & (y < 3);
    square = 6;
elseif (x < -3) & (y < -3);
    square = 7;
elseif (x > -3) & (x < 3) & (y < -3);
    square = 8;
elseif (x > 3) & (y < -3);
    square = 9;
end;

Figure 7. Function turn() to convert mouse click to square number

observe students successfully debugging branches of the code to which they did not contribute while failing to detect certain errors in their own branch. This provided valuable illustration to students that the author of a program can often be its least effective tester. Lacking an enforced methodical testing process, the author of a program is more likely to subconsciously stay within certain operating boundaries unrecognized by someone not familiar with the program. In this case, students failed to explore every possible game scenario of their own code but seemed far more attentive when dealing with the sections written by others. Once the game was fully debugged, the program was evaluated (played over and over and over) to determine if it met its original design objectives.

Figure 8 displays the opening screen before the user’s first move is entered. Note the large crosshairs denoting the current mouse cursor positioned in square 1. Left clicking at this position will cause an X to be drawn there. In Figure 9, the game has progressed to include the computer’s first reply move in square 5, followed by the user’s second move in square 2, and then the computer’s reply move in square 3. The board is shown awaiting the user’s third move.
Figure 8. Game board prior to first move, mouse cursor (crosshairs) over square 1

Figure 9. Game in progress, moves 1-5-2-3
The example game continues with the user next moving in square 7 followed by the computer’s move in square 4. On the next move, the user clicks on square 8. Figure 10 shows the board in the split second after the user’s move is entered before the computer’s next move has been displayed. The program logic operates so quickly that the computer can display its reply move immediately after the user’s move is registered. To the user, it appears that both moves are entered at the same time. This was deemed to be unpleasant for the user. The computer must at least appear to be “thinking” for some brief period in order to retain the player’s interest in the game, so the draw_o() function included a pause() statement and a red “My turn” banner to generate this effect for a fraction of a second. Although most programs are designed for the fastest possible execution speed, this intentional slowdown to produce more compatible user interaction was another important illustration for students that true program optimization is more than just minimizing execution speed. Sometimes, an unconventional approach is required.

![Figure 10. Game in progress, computer “thinking”, moves 1-5-2-3-7-4-8](image)

Inspection of Figure 10 should reveal the user’s obvious error; by failing to block the computer’s play across the center row, the user is about to lose the game. As expected, the computer completes that row, indicates its victory, and prompts the user to play again or quit the game (Figure 11). Students should be queried as to why clicking on the center square and clicking anywhere else on the board should be used for the continue/quit input. Those on the ball will realize that the center square is the only one that is not repositioned during any of the rotations necessary for the reduced-scenario algorithm (see Figure 4), so clicking on that square will always return the same square number. Using another if-else block, this input was used to
reset the board and play again, with any input other than the center square terminating the program.

![Figure 11. Game over (computer wins), moves 1-5-2-3-7-4-8-6](image)

When the user does not make a move that allows the computer to win, the game continues until the players draw. Figure 12 illustrates such a game. Based on the instructor’s problem assessment that the computer will never losing a game, there was no function `lose()` written as the counterpart to `win()`.

As mentioned earlier, and for obvious reasons, the user must not be allowed to move into an occupied square. The `if-elseif-else` blocks in the student functions are ideally suited to trap for this error condition. If all of the legal moves are handled in the `if` and `elseif` statements, all that remains are the illegal possibilities. The function `illegal()` was provided to display an error message to the user (see Figure 13) and to redirect program control back to the calling function to permit a legal move to be made. This function serves a dual purpose. Not only does it further implement modular style, but it also circumvents the need to introduce the technique of recursion within the scope of this project (although the term was introduced as a foreshadowing of things to come). By passing control to `illegal()`, the calling function may once again be invoked using Matlab’s `run()` function without the need for a recursive function call. As with any program, there are always alternate programming techniques to accomplish the same objectives. Those used in this project were judged to be the most beneficial for introductory programming students even when there were more elegant or terse options. The emphasis in

“Proceedings of the 2004 American Society for Engineering Education Annual Conference & Exposition
Copyright © 2004, American Society for Engineering Education”
Figure 12. Game conclusion (draw game), moves 5-1-3-7-4-6-2-8

Figure 13. Error detected, moves 6-5-6 (square not available)
this course resides in introductory algorithm conceptualization and implementation. Students will be able to focus on the finer aspects of the art of programming in subsequent courses.

The Student’s Course Evaluation and Subsequent Progress

As anticipated, all but one of the students who enrolled in the course came from CS1. Some elected to remain in CS1 and added this class to their existing schedules, while others used the drop-add process to substitute this course for CS1. One student who had successfully completed CS1 enrolled in the course to acquire proficiency in Matlab.

The initial offering of the course was very well-received by everyone enrolled. On the 1-5 (low to high) evaluation scale used by the institution, the course received ratings of 4.7 for course and instructor excellence. Reinforcing their approval, typical comments from students included “excellent course” and “I thought this course was great. It gave me another aspect to thinking about programming and rekindled my interest to program.” In the future, students will be asked to provide course evaluation information beyond that collected by the institution in an attempt to further improve content and delivery. Additionally, when sufficient data has been obtained, it will be used by college’s academic assessment committee as the basis for as assessment report on student retention in the engineering and engineering technology degree programs.

Two-thirds of the students from the first offering last semester have re-enrolled in CS1 this semester and are, by all accounts, faring much better. With half of the CS1 course remaining, it is still too early to reach a final determination, but the level of their work is now well beyond the median level of the rest of the class. The remaining third of the CS 103 class has indicated their intention to re-enroll in CS1 next fall (scheduling conflicts were cited as the reason for their not enrolling this semester). Accordingly, and as confirmed by their own course evaluations, the preliminary conclusion is that the first cohort of students have benefited from their participation in CS 103.

The Instructor’s Course Evaluation

With only one section completed and another underway at the time of this writing, quantitative outcome assessment beyond that provided above is not yet available. This is clearly a work in progress. However, the experience to date certainly permits a preliminary qualitative evaluation.

The course was very enjoyable to teach, especially for a first offering. The instructor felt that the objectives were largely met, but much was learned that will improve subsequent offerings (see below). Students seemed to relish the opportunity to work on a very atypical problem for an “engineering programming” course. Class discussions were lively with participation from all students to an acceptable degree. The quality of the students’ code contributions varied quite a bit. While every student contributed some fully-functional code, some had much greater difficulty locating faults and successfully debugging their code. Providing this experience within a semi-relaxed group environment was a primary goal of the course, and the instructor believes that the lessons learned will benefit the students in their further studies.
Several students had difficulty starting their work, probably due to the familiar “inertial effect” (work on a project is sometimes hard to begin but easier to sustain once the student gets rolling). Supplying the initial control function (to select the correct main program branch from the three possibilities) was important to provide a format example and a defined starting point. From that point forward, student progress was self-sustaining; the more they accomplished, the faster they progressed. The entire prototype class passed the course with a uniform distribution of passing grades (As, Bs, and Cs).

Tic-tac-toe provided, in the instructor’s opinion, and excellent vehicle for this course project. It was possible to illustrate a myriad of programming techniques and practices without the need to generate overly-complex code. By providing a framework within which students could focus on a series of small and clearly-defined operations, the trepidation usually associated with writing an entire program from start to finish was overcome. A group environment allowed students to help each other, fostered teamwork, and simulated (to a certain extent) the dynamics that they are likely to encounter in the workplace. The satisfaction of playing an entertaining game program to which they contributed provides an unmistakable boost to their confidence and increases their motivation to continue on the road toward a technical degree.

Finally, the students’ question of the instructor’s assessment of the game of tic-tac-toe was resolved in his favor by the students themselves. Once the program was finished and fully debugged, no one was able to defeat the computer by overcoming its unfailing ability to always make the best possible move. No longer was it necessary for students to question the suitability of the algorithm; they were able to exercise it to their heart’s content until convinced that it could not be beaten. Nothing could have been more persuasive.

Suggested Improvements and Future Course Directions

The late start date of the course is essential in meeting its objective of providing a drop-add alternative for struggling CS1 students. However, the pace of the course must take the late start into account. While the introductory exercises are necessary to build familiarity with Matlab, it is easy to dwell too long on this material at the expense of time remaining for the project. In the next section of this course, the class project will be introduced earlier in the semester, well before students are ready to begin writing code. They will be able to evaluate the project requirements and begin considering algorithm possibilities while they are still acquiring the basic programming skills necessary for the course. Additionally, the pre-project exercises will be fewer in number and more specifically directed to that semester’s project so that all of the fruits of those exercises can be used in the project.

In spite of its excellent suitability, the course can not rely on this particular tic-tac-toe project each semester. A series of different programming projects of the same nature must be developed to keep the course fresh. For the coming semester, the class project will begin with the tic-tac-toe code suite developed in the inaugural section and expand it to enable a first move by the computer. The method of selecting which player will move first (random, alternating, last winner, or user-selected) will be decided by the students during program development based on their assessment of game play. Although the basic nature of the project will remain unchanged,
an entirely new series of student-written functions will be necessary to implement this feature. It will also allow the introduction of new programming techniques (including randomization, so the computer doesn’t always make the same first move). The following semester might develop a third-generation version of the program to permit the computer to play against itself, as this would represent the confluence of the work developed by the first two sections with additional control features to be added that semester. Following this sequence of tic-tac-toe projects, at least one similar sequence in another fun but challenging programming topic should be developed and utilized before returning to the tic-tac-toe sequence in future years.

Due to its fundamental reliance on matrix-based techniques, Matlab would also provide an excellent “half-step” to the CS1 image processing instructional module developed by UNR’s Computer Vision program. This module serves as a course capstone in C++ programming courses but requires significant program overhead (including extensive file I/O) to accomplish. Student time devoted to writing and perfecting this overhead, while valuable in its own right, dilutes the time available for algorithm development. By simplifying the required support tasks, the use of Matlab in lieu of C++ will permit instructors with limited time to concentrate on the image processing functions instead of the program overhead. At this level, anything that can be realized in a high level programming language can also be implemented in Matlab with greater ease. Following an introduction to image processing in this manner, full program development can remain the focus of a subsequent CS1 course using the Computer Vision module.

This time around, the instructor contributed a moderate amount of code to the project. It would be preferable if all code was generated by students at some level. The class, by design, largely populated by students requiring a gentle introduction to programming. A number of the ancillary features required by the project are more complex than the control functions that these students may normally be expected to write and are therefore beyond their present horizon. For that reason, it would be desirable to attract at least one or two students who would bring more advanced programming abilities to the class and be able to provide these ancillary functions. As this course can not be applied toward as an elective toward an Engineering Technology or Engineering Science degree (only as a technical elective in non-technical degree programs), there is unfortunately little motivation for such students to enroll. On occasion, another student might like to learn Matlab in this environment and would not be concerned about the lack of applicability toward his/her technical degree. However, an advanced programming student seeking an honors project or some independent study credit might fill this role instead. The course might also be team-taught with a CS graduate student seeking teaching experience.

As discussed above, this course was taught in a Linux-based computer laboratory. While the author militantly advocates Linux proficiency for all Engineering, Engineering Technology, and Computer Science students, providing an introduction to this operating system consumed valuable course time and rudimentary shell operations were the source of many questions throughout the semester. Some time might be saved by foregoing the added value of Linux exposure at this point in the program in favor of using the Microsoft Windows® environment, but there would certainly still be students who require help with basic operations there as well. In a true entry-level course, students can not be expected to be fully proficient in this regard. Future sections of CS 103 will likely be supplemented by a weekly “open lab” session for drop-in help. Students not yet comfortable with computer basics will be directed to this session for further skill
development, saving valuable class time. If the course is taught under Linux, it is strongly urged that students be provided with lab access beyond scheduled classes for study and practice.

Ideally, students should also have the option to acquire the software used in the course to install on their personal machines. Matlab offers a fairly economical student version that can be ordered for sale in the school’s bookstore.\textsuperscript{10} Due to the cost, this was not required in this course but was presented as an option. If an open-source or GPL program is chosen in lieu of Matlab, the source code and binaries could be freely available to all on the internet.

**Bibliography**

1. http://www.wncc.edu
3. http://coeweb.enr.unr.edu/

**Biographical Information**

DAVE WILLIAMS is the Engineering Instructor at Western Nevada Community College in Carson City, Nevada and a collaborator in Computer Vision research and development projects at other institutions. He is a Registered Professional Engineer and has practiced as a consulting engineer for 30 years in the field of antenna design and analysis, computer simulation and modeling, and electromagnetic compatibility. Holds the Ph.D. degree in Electrical Engineering from the University of Nevada, Reno.

DWIGHT EGBERT received his Ph.D. in Electrical Engineering from the University of Kansas in 1976. He is in his seventeenth year as a full-time academic engineer with the University of Nevada, Reno College of Engineering and is currently the Director of Computer Engineering Undergraduate Programs and Director of the College of Engineering Computer Center. He has authored more than 80 original research papers and reports dealing with computer applications in computer vision, remote sensing, image processing, and neurocomputing.