6-18-2006

Effective Teaching of Programming Concepts Using Low-Resolution (Character) Graphics

Tamisra Haran Sanyal
University of Dayton, tsanyal1@udayton.edu

2006-2025: EFFECTIVE TEACHING OF PROGRAMMING CONCEPTS USING LOW RESOLUTION (CHARACTER) GRAPHICS

Tamisra Sanyal, University of Cincinnati

Tamisra H. Sanyal holds the position of Assistant Professor in the Department of Information Technology, College of Applied Science, University of Cincinnati. Prior to joining his present position he held teaching positions in Penn State Altoona College and in Monroe Community College (Rochester, NY). He has taught introductory and advanced programming courses in a variety of high level languages as well as courses in Data Structures, Networking, Computer applications, Unix, Discrete Mathematics, Linear Algebra, and Physics. He is interested in working to improve CS/IT teaching and learning and has published in this area.

Effective Teaching of Programming Concepts using Low Resolution (Character) Graphics

Introduction

For novice students of Computer Programming, the difference between Floating point numbers and Integers (and consequent difference between arithmetic operations on them) appears to be a difficult concept to grasp. In popular languages such as C++ or Java, this difficulty is made more acute by the fact that the same symbol is used for both floating point division and integer division.[1] Without a proper grasp of the difference between these two fundamentally different kinds of numbers, students would also find it difficult to understand that floating point results should not be compared for equality. I was therefore interested in finding a way to reinforce these concepts soon after the students were initially exposed to them.

Nested Loops and Character Graphics

Drawing some simple shapes like rectangles or triangles in a text window can be an interesting application of nested loops and introductory programming textbooks often include such examples.[2] Drawing shapes like a circle is usually not considered unless a high resolution graphic window is to be the output device. I felt that a programming exercise to draw a circle using a text window can be useful because:

- The exercise can be considered as soon as the students are exposed to the concepts of nested loops,
- Knowledge of a graphics package is not required,
- The exercise involves significant problem solving activity.

To use this exercise for reinforcing the concepts mentioned in the introduction, we would first consider drawing a filled circle, and then consider drawing a circle in outline. The failure of the program to work in the second case provides the basis for a classroom discussion on the differences between floating point and integer arithmetic.

Problem Solution

We consider the output device (a printer using a fixed pitch font or a text window) to be capable of displaying a certain number of rows of characters. For example, a standard text window may have 25 rows of 80 characters each. The top left position of this grid has coordinate (1, 1). In this case, (40, 13) is the coordinate of the center of the screen (not using the column position 80). The (column, row) positions are translated to (x, y) coordinates relative to the center of the output window using characters per inch (CPI) and lines per inch (LPI) values for the particular output device. To draw a filled circle, all character positions satisfying $x^2 + y^2 < r^2$ display a non-blank character whereas the rest display the space character. The essential nested loop is given in the following figure. Note that most of the named constant values are determined by the output device characteristics. The only input provided is the radius of the circle to be drawn (r).

```
      // nested loop to go through all character positions
      for (int j = YLOW; j <= YHIGH; j ++) {    // for each line
            // calculate y coordinate relative to center of screen
            y = (j - YCENTER) / LPI;

            for (int i = XLOW; i <= XHIGH; i ++) {
                  // for each character position within line
                  // calculate the x coordinate relative to center of screen
                  x = (i - XCENTER) / CPI;

                  if (x * x + y * y < r * r)
                        cout << '*';
                  else
                        cout << ' ';
            } // end for i

            cout << endl;     // end one line
      } // end for j
```

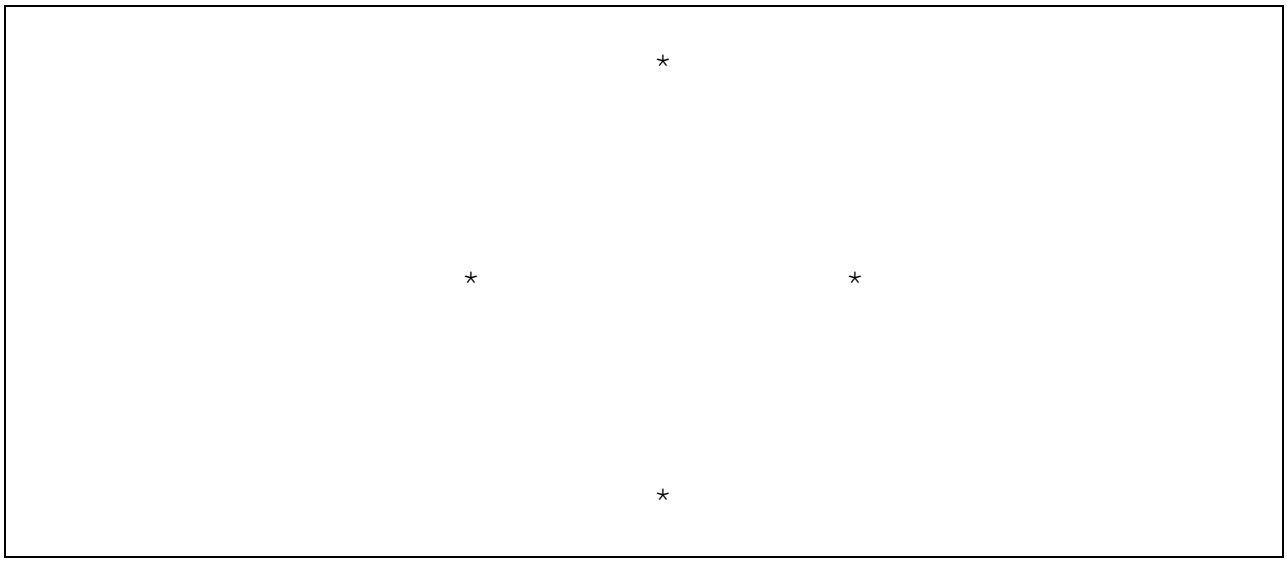
Results from this exercise

If the LPI and CPI values are correct for the output device, then the program draws a shape that does look like a filled circle. The following figure shows a sample output from this exercise. Note that the exact values for character pitch and line spacing is essential for correct output and using inaccurate values result in some distortion.

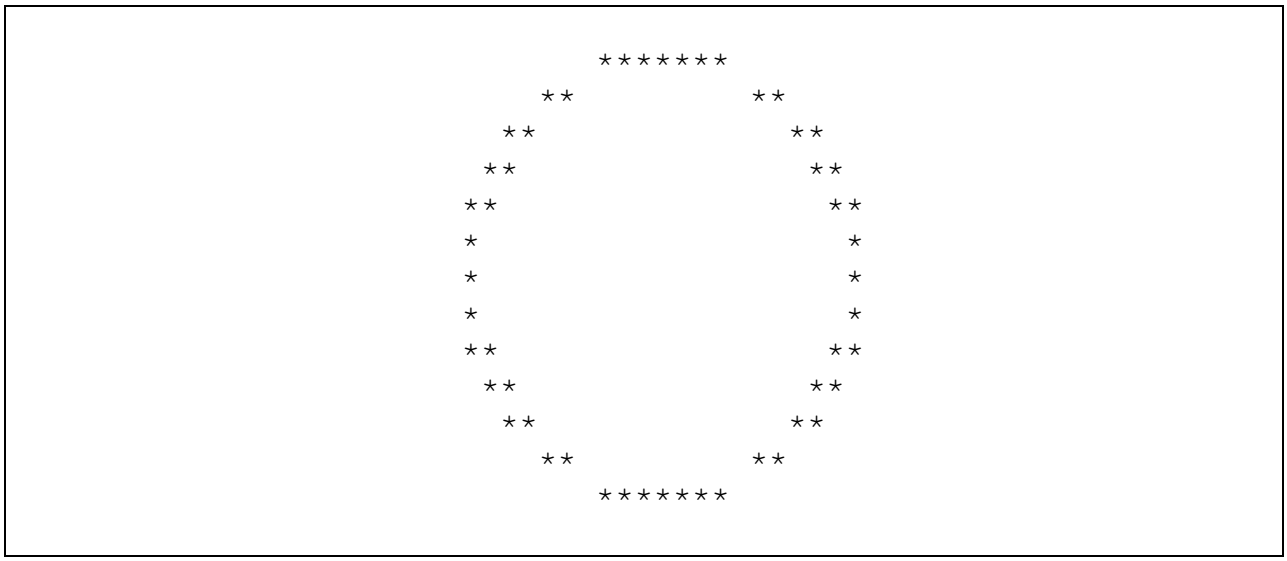
```
                    *********
                  *************
                 ***************
                ******************
                ******************
               ********************
                ******************
                ******************
                 ***************
                  *************
                    *********
```

Drawing the circle in outline

The next step in this exercise then is to consider drawing the shape of the circle in outline. At this point, the obvious change is to replace the < comparison with equality comparison (i.e. replace the logical expression **`x * x + y * y < r * r`** with **`x * x + y * y == r * r`** in the **`if`** statement inside the nested loop). This change is made and the output shows only four points on the circle where x = 0 or y = 0. The sample output from this change is in the following figure.



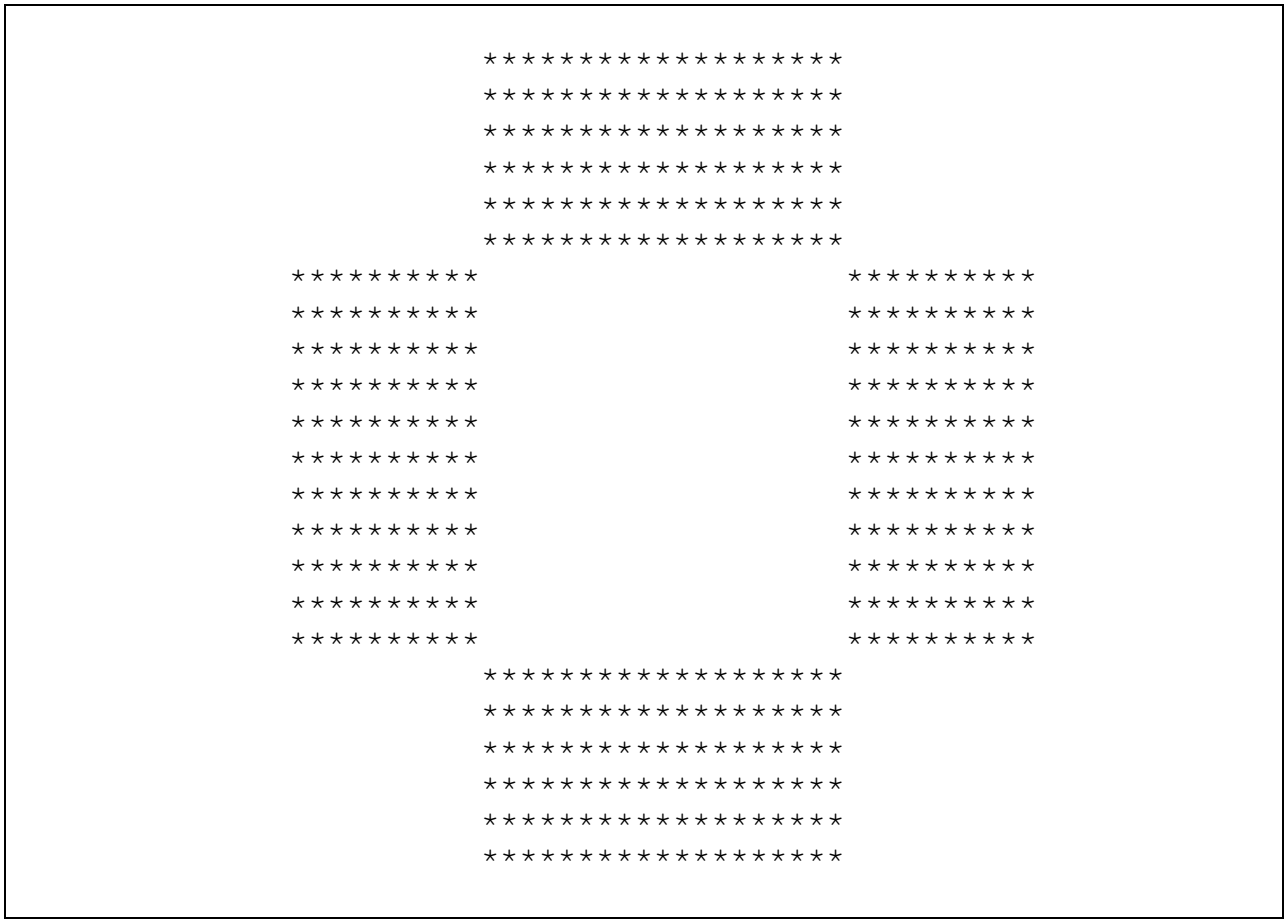
At this point I remind the students that we cannot expect the (column, row) grid positions of our text window to satisfy the equation of the circle exactly and so we must test for approximate equality. By introducing an appropriately valued tolerance parameter we then modify this program to work correctly. The logical expression **`x * x + y * y == r * r`** is now replaced by: **`fabs(x * x + y * y - r * r) <= tolerance.`**
The resulting output is shown below (for suitable values of *r* and *tolerance*)



Highlighting difference between integer and floating point arithmetic

I then introduce a fourth version of the program which is identical to the third version except that the named constants LPI and CPI are declared as integer numbers rather than floating point numbers. For standard line printers LPI = 6 and CPI = 10 so it seems reasonable to declare these as integer type rather than a floating point type. This causes the translation from (column, row)

positions to (x, y) coordinates to use integer division and as a result the circle morphs into a collection of rectangles or almost disappears (depending on the value of the radius chosen). See a sample output below.



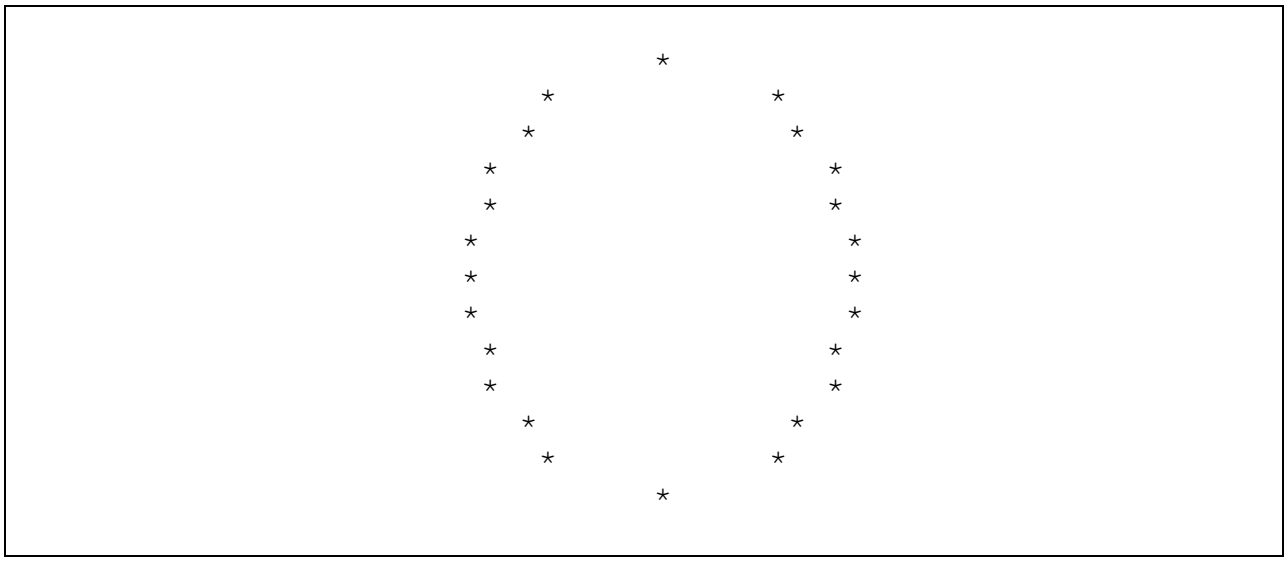
This dramatic change in output tends to draw the attention of the students and I take the opportunity to point out the difference between integer and floating point division.

Efficiency issues

All of the versions discussed above have the same basic nested loop structure where for each character position in the output device, we make a determination whether to put a blank space or a visible character there. A more efficient solution is possible if for each line position we calculate the corresponding *y*-coordinate, and then solve the equation of the circle for *x* (i.e., $x = \sqrt{(r^2 - y^2)}$) to find the corresponding *x*-coordinates, and plot the points on the output device. This solution now requires only one loop, and can be used as an illustration of how more efficient solutions can be devised by further analysis of a problem. The following figures show the essential modification to the problem solution and a sample output.

```
// loop to go through all lines in the window
for (int j = YLOW; j <= YHIGH; j ++) {    // for each line
      // calculate y coordinate relative to center of screen
      y = (j - YCENTER) / LPI;
      float ysq = y * y;            // square of y
      if (ysq <= rsq) { // no solution otherwise [ rsq = r * r ]
            x = sqrt(rsq - ysq);          // the positive root
            // x coordinates of two grid points that satisfy
            // the equation (closest possible)
            int first = round(XCENTER - x * CPI);
            int second = round(XCENTER + x * CPI);
            // plot first point
            cout << setw(first) << '*';
            // plot second point only for non-degenerate case
            if (second > first)
                  cout << setw(second - first) << '*';
      } // end if
      cout << endl;           // end one line
} // end for j
```

Sample output



One notable difference between this picture and the one with the nested loop is that in each line only one or two points are plotted and so one may consider this output to be visually less appealing.

Conclusion

I have used this particular exercise sequence in my classes a few times and it certainly aroused my students' interest to see small changes in a program lead to unexpected results. The exercises described can be implemented in major languages in use today with little modification. I believe that the examples given here show that Low resolution graphics can be exploited to enhance students' appreciation of some aspects of computer programming that are not easily assimilated.

While today's computers generally provide for bit-mapped (i.e., high resolution) output devices, the packages to be used tend to be platform dependent. A notable exception is the Java language in much use today, which does provide for a graphics package that can run on any platform. However, I should note that learning to use such packages (at least in Introductory Programming classes) usually involves instantiating some objects from a library class and calling the required methods, and do not require any significant problem solving activity. Of course, dealing with individual pixel positions to draw shapes would require solving a problem, but the essential features of such problem solving can be tackled even in the low resolution context. For this reason I think that low resolution graphics should be more widely used to create interesting programming exercises as the solution techniques are essentially independent of computing platform or programming language that one may be required to use.

Bibliography

1. Programming and Problem Solving with C++ (2nd edition) (Instructor's Guide p. 28), Nell Dale, Chip Weems, and Mark Headinton; Jones and Bartlett.
2. C++ How to Program (4th edition) p. 165, H.M. Deitel and P. J. Deitel; Prentice Hall.