Assignment 1

CS6533/CS4533 Spring 2015

Due: Wed. 2/25 in class

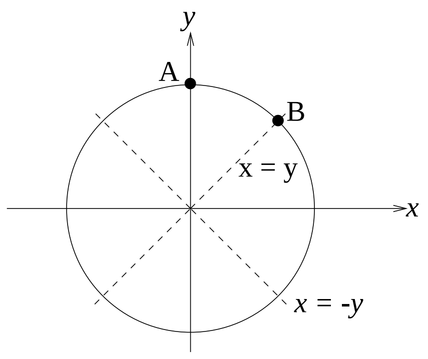
Note: This assignment has 3 pages; total score: 120
This assignment is to implement the Bresenham's scan-conversion algorithm for **circles** and some simple transformations, as well as a simple animation to grow the circles, using the OpenGL programming libraries under Visual C++.

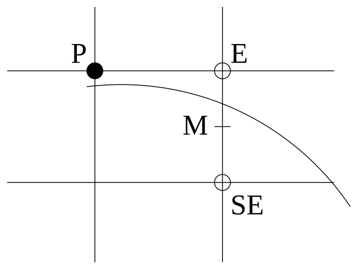
General Instructions (as described in the syllabus): Submit your write-up, your source code (with full comments and documentation), and include your e-mail address as well as brief instructions on how to compile and run your programs. Submit everything by sending an email to the TA (if your write-up is hand-written, scan it into a PDF file to be sent by email).

(a) (This part only involves a write-up rather than a programming.)
In class we described the Bresenham's scan-conversion algorithm for a line segment; your task here is to generalize the ideas and derive Bresenham's scan-conversion algorithm for a *circle*. Again, the main goal is to avoid all floating-point computations and to involve only integer computations.

To start, assume that the circle in question is centered at the origin $(0, 0)$ with radius r where r is an *integer*, i.e., the circle is given by $x^2 + y^2 = r^2$. Consider splitting the circle with the two axes and the two lines $x = y$, $x = -y$ into eight symmetric regions (see the left figure below). We only need to consider the region from $A = (0, r)$ to $B = (r/\sqrt{2}, r/\sqrt{2})$, assuming that there is a function **circlePoint(x, y)** which, for each point (x, y) in the region from A to B, draws the eight symmetric points in the eight regions corresponding to (x, y) (it is a simple task for you to find out these eight points).

The overall strategy is as follows. Starting from A, in each iteration we increase the x-value by 1 and choose the best y-value, until B is reached (or more precisely, until B is **about to be passed**). Suppose currently $P = (x_p, y_p)$ is chosen; the two possible candidates next are E and SE (see the right figure below), and $M = (x_p + 1, y_p - 1/2)$ is the midpoint between E and SE. To decide which candidate to pick, we let $F(x, y) = x^2 + y^2 - r^2$ and define a decision variable $D = F(M)$. If $D < 0$ then M is inside the circle (as the case shown in the figure) or otherwise M is on or outside the circle — the correct candidate can then be chosen accordingly.





Describe which candidate to pick in each of the two cases, and derive the computational formulas for D, including the one for computing D_{start} and the ones for incremental updates (computing D_{new} from D_{old}, distinguishing two cases depending on the sign of D_{old}).
(**Hint:** The derived result for D_{start} may involve a non-integer term but you can easily replace it with an integer term so that the resulting D_{start} still makes the same decisions in selecting E or SE.)
Turn in: A write-up as required above. **(30 points)**

(b) Copy a file from `http://cse.poly.edu/cs653/assg1/sample.c`. Compile and run it. A window should show up with a tiny yellow point. This point is drawn via the `glVertex2i()` function call. Change the two arguments of this function to draw some more pixels to get an idea of the screen coordinate system used by OpenGL.
Turn in: A description of the screen coordinate system used by OpenGL (i.e. origin and orientation of the x and y axes) relative to your window. Can you display pixels at (x, y) where either $x < 0$ or $y < 0$? **(5 points)**

(c) Implement a routine `draw_circle(int x, int y, int r)` that will draw a circle centered at (x, y) with radius r, specified in the screen coordinates. You should implement the Bresenham's algorithm derived in **(a)**, together with an *enhanced* version of **circlePoint()** so that the circle center (x, y) is **not** restricted to the origin. Modify the code from **(b)**, in particular, use the `glVertex2i()` function to draw your pixels (and change the point size to 1.0 by using `glPointSize(1.0)` in the code from **(b)**). **No points will be given if you use any OpenGL circle drawing functions.** You should avoid any floating-point computation. To test, your code should ask the user to enter the three integers from keyboard and call the routine to draw the circle.
Turn in: We will directly look at your source code and run your program. **(40 points)**

(d) Add a file-input ability to your code in **(c)** so that it will read an input file in the format:

$$\begin{gathered} n \text{ (meaning: number of circles)} \\ x_1\ y_1\ r_1 \\ x_2\ y_2\ r_2 \\ \vdots \\ x_n\ y_n\ r_n \end{gathered}$$

where all x_i, y_i, r_i values are integers but *arbitrary*, positive or negative (except that r_i is always positive). A sample input file, `input_circles`, is provided in the web site `http://cse.poly.edu/cs653/assg1/`. Store the values read into some appropriate data structure. Your task here is to draw the n circles specified by the input file by calling `draw_circle()`, where the x_i, y_i, r_i values in the input file are in the **world coordinates**. You should perform a suitable world-coordinates-to-screen-coordinates transformation so that

1. the origin of the world coordinates maps to the center of the OpenGL graphics window, and the x-axis increases to the right and the y-axis increases to the top; and

2. the scale of the x- and y-axes should automatically adjust so that the OpenGL window accommodates all points of the input circles and that the shape of the circles drawn is not

changed (i.e., they are still circles and not ellipses, etc).

Note that the size of the OpenGL window should stay fixed. You should first perform the transformation on the input x_i, y_i, r_i values and then use the resulting integer screen-coordinate values to call `draw_circle()` of **(c)**. Incorporate this part as a new option in your code.
(**Hint:** Compute a uniform scaling factor for both x- and y-dimensions from the input file.)
Turn-in: We will directly look at your source code and run your program. **(25 points)**

(e) This part builds on top of **(d)** to develop a simple animation of growing circles. Your program should read the sample input file `input_circles` mentioned in **(d)** and use your result of **(d)** as the final frame in the animation cycle—suppose that there are t circles and each circle i has radius r_i in this final frame (r_i has been computed from **(d)**). Your task here is to add an animation capability so that in K frames, each circle i has its center unchanged but grows its radius from $r_i/K, (r_i * 2)/K$, and so on, until r_i (i.e., in the j-th frame, each circle i has radius $(r_i * j)/K$, for $j = 1, 2, \cdots, K$), and then this animation cycle is repeated again, forever (until the user closes the graphics window). Choose an appropriate K value to make the circles grow smoothly and in a reasonable speed. **(20 points)**

Note: For the programming parts, turn in a single program that incorporates Parts (c), (d) and (e) as options for the user to choose.