Computational Geometry
Fall 2013
Project #1
Due: October 21

General Description

In this project you are expected to "play" with some consequences that the integer-only representation imposes on different geometric problems. Specifically, you will be able to get a hands-on experience of what it really means to display something on a screen – things that we take for granted in our everyday life…

Most of the algorithmic aspects needed for this project have either been discussed in class, or explanations/hints are provided to you (see below) or, in the worst case – plenty of implementations that can help you are available, so please feel free to use them to whatever extent you deem appropriate (just don't steal and carbon-copy an entire implementation done by someone else).

Needless to say, the submitted programs need to exhibit some good code-organization properties (e.g., indentations, meaningful mnemonics used for variables, comments), however, in addition to it (for extra credits – just to provide motivational incentive) you are expected to address some elementary issue of robustness (e.g., your implementation should not crash if an invalid type of input is presented).

Background

We now present some preliminaries

Bresenham Algorithm

As discussed in class, one of your tasks is to implement an algorithm that will display an elementary 2D geometric object – line-segment, on a screen.

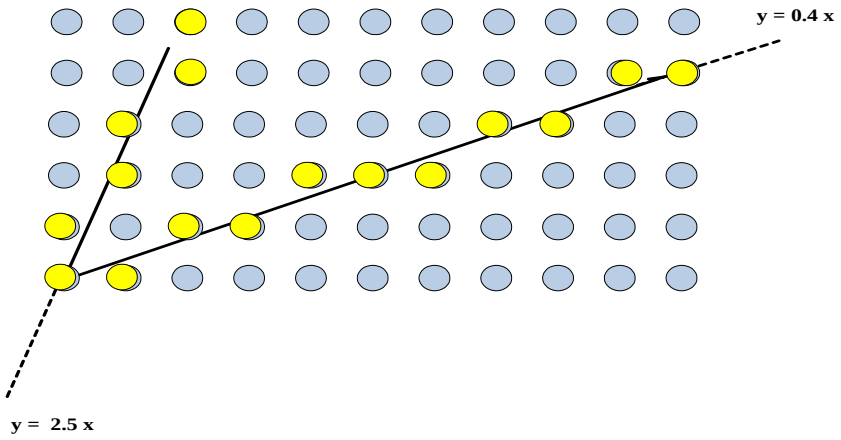

Fig. 1 Highlighting pixels.

Recall the example: Assume that we are given the line with equation Y = 0.4X and that the bottom-left disk coincides with the origin of the coordinate system. A unit-change (increment) in X changes the value of Y by 2.5. However, since we are restricted to integer-values, one may note that we can "semi-bypass"

the multiplication by:

$$Y_{i+1} = 0.4\ X_{i+1} = 0.4(X_i + \Delta X) = Y_i + 0.4\Delta X$$

Hence, when $\Delta X = 1$, we have $Y_{i+1} = Y_i + 0.4$ (recall that we know $X_{i+1} = X_i + 1$)
This is at the heart of the, so called, *Incremental Scan-Conversion algorithm.* At each step, we increment X and we check to which integer value should the actual value of the variable Y be rounded-to.
However, this approach requires rounding which, back in the days, used to take some time… To alleviate this, J.E. Bresehnam published the famous *Algorithm for Computer Control of a Digital Plotter*, (IBM Systems Journal, 4(1), 1965) which is based on the so-called *midpointtest.*
An illustration is provided in Figure 2 below.
Assume that we have just computed the previous pixel (e.g., in the second point on the Figure, (X2,Y2)). Now, at X3, we must choose between thebottom pixel (commonly called "East") and the top pixel ("North-East").
One can readily argue that the same effect as the basic Scan-line algorithm can be achieved if the vertical distance between Y3 and E is compared to the vertical distance between Y3 and NE. However, an equivalent formulation would be: ***detect on which side of the line does the midpoint M belong, and select the other pixel.*** The main benefit of this observation is that now, one can define a decision-variable based on the relationship with the midpoint between two vertical-neighouring pixels. Essentially, let F(X,Y): AX + By +C = 0 denote an implicit form of the equation of a line
(the values can be readily obtained if an equation is given in, e.g., slope-intercept form). Now, at the midpoint M in Figure 2 we have that F(M) = F(X2 + 1, Y2 + ½). However, X2 and Y2 were the values in the previous "iteration". Hence, the decision variable DV can be defined as DV = A(Xp+1) + B(Yp + ½) + C (where the index "p" stands for "previous").

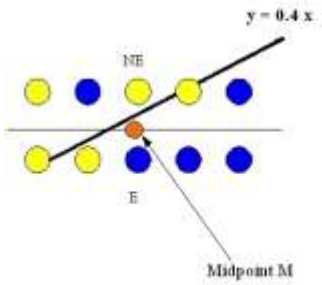

Figure 2. Midpoint

As a rule, if DV > 0, we highlight the pixel "NE", and vice-versa. Now, a straightforward math can show that $DV_{new} = DV_{old} + A + B$, which provides a methodology of updating the decision variable as we move along the X-axis. Note that now, all the
operations needed are reduced to simple additions, instead of multiplications, additions and round-offs. Clearly, there are some degeneracies (e.g., horizontal lines) that can be handled straightforwardly, however, an important observation is that the updates of (and the logic behind) the decision-variables will vary depending on whether the slope of a given line is: (1) positive vs. negative; (2) > 1 vs. ≤ 1 (hence, 4 cases total).

Flood-fill Algorithm

In computer graphics, a *region* is defined as a *collection of pixels*, and there are two basic types: (1) 4-connected, which is, every 2 pixels can be connected using only up/down and left/right moves among neighboring pixels; (2) 8-connected, which is, every 2 pixels can be joined by an additional (with respect to 4-connected) "jumps" of up-and-left, up-and-right; down-and-right; and down-and-left. Assuming that some default coloring of the pixels is available, a region can be defined in two ways: (1) interior-defined, which is, the largest connected region whose pixels have a same color as a selected pixel from the interior; and (2) boundary-defines, which is, the largets connected region of pixels whose color-value is not equal to some given boundary-color-value. In *flood-fill approaches*, it is common to assume that the regions are interior-defined.
In the (good?)old days, the flood-fill approaches were considered problematic not only because their

recursive nature implied time-consumption, but also because they used to cause stack-overflow when the limits on the memory are small. Hence, more efficient region-filling algorithms were developed, based on (so called) spans – which are horizontal scan-lines detecting a boundary-pixel and marching towards the interior.

Given the integer coordinates of a point in the interior of a region (for this assignment, assume only convex polygons), and assuming: (1) known OldColor of the pixel color; (2) known NewColor of the desired flood-fill color; (3) known color of the boundary (edges), the basic recursive FloodFill algorithm has the following steps:

begin
ColorPixel(x,y, NewColor);
FloodFill(x, y-1, OldColor, NewColor);
FloodFill(x, y+1, OldColor, NewColor);
FloodFill(x-1, y, OldColor, NewColor);
FloodFill(x+1, y, OldColor, NewColor);
end

Clearly, prior to entering the block an exit-condition is needed…

Tasks

For this particular project, you are at complete liberty to use need to use an IDE of your choice (Java-based; C++ based; *processing* (available at *processing.org*) and implement algorithms that will achieve the following:

1. Given a pair of integer coordinates denoting the end-points of a given line segment, display that segment by highlighting the corresponding pixels.
2. Given a polygon, represented by the sequence of its vertices in counter-clockwise order, highlight is border (pixels corresponding to the edges) and color the interior of the polygon with a different color.
3. Given two polygons, determine their intersection and color it.
4. Given a sequence of points (assume that they will already be ordered by their X-coordinate), display their convex hull

Extra-Credit: Given a polygon and a line-segment, display the portion of the line-segment that is in the interior of that polygon (this is commonly called line-clipping in computer graphics, and it's used when representing entities on a display with limited dimensions).

As indicated above, you may assume (except for task#4) that the polygons will be convex. You are at complete liberty to "shop" around for code samples, however, the source code that you will be submitting, which will be tested in the processing, must be yours. Note that you have already been given some hints regarding sites where most of the stuff needed can be readily available.

Notes + TurnIn:

You are at a complete liberty to look at corresponding algorithms at any source (website, or otherwise). For that matter, you are urged to take a look at the CGAL (Computational Geometry Algorithms Library) available at:

http://www.cgal.org/Manual/latest/doc_html/cgal_manual/contents.html

with a bunch of well-documented libraries (do so, regardless of your IDE-choice…).

You may assume that the inputs will be given in a plain-ASCII file, with the following format:

1. The first line will contain:
   - an entry of the form (L), where the character "L" denotes that (the values of) the rest of the inputs (two of them) should be interpreted as coordinates of the points denoting the endpoints of the line segment to be displayed (one point per line, comma separated coordinates); OR

- an entry of the form: (P), where the character "P" denotes that (the values of) the rest of the input pertain to a polygon/region – again, one point per line and you may assume that the sequence will represent the vertices in a counter-clockwise order of traversing the polygon.

NOTE: you are more than welcome to make everything fully-interactive (i.e. not only can the user enter the respective file name(s) but, should he chooses to do so, he can enter the values one-at-a-time).

A separate submission-folder will be created under the "Assignments" on the Blackboard and, upon completion, you submission should consist of a zip-ed folder that will contain:

1. All and only the source files for each of the tasks;
2. A "ReadMe" file that will describe any specific aspects that you feel that a grader of your work (e.g., compilation requirements/paths, any known bugs still present at the time of submission, etc…).

Start working ASAP.

Good luck.