H2q1

Directory: export/home/cs450/cs450119/Hwork2

Run with: java h2q1

To calculate the distance between the clicks, first stored the value of the two click’s evt.getX()/evt.getY() into the variables:

(click1X, click1y), (click2X, click2Y)

Next, to calculate the approx. distance between the clicks:

Then rounded the distance to the nearest integer to get the approximate distance in pixels. This is implemented using the java pow() and sqrt().

**Testing**:

|  |  |  |
| --- | --- | --- |
| click 1: (12,18) click 2: (636,19) approximate distance: 624  click 1: (163,111) click 2: (169,110) approximate distance: 6  click 1: (213,182) click 2: (215,182) approximate distance: 2  click 1: (153,39) click 2: (151,124) approximate distance: 85 | click 1: (70,106) click 2: (81,104) approximate distance: 11  click 1: (126,101) click 2: (161,112) approximate distance: 36  click 1: (93,37) click 2: (110,37) approximate distance: 17  click 1: (111,37) click 2: (155,36) approximate distance: 44 | click 1: (210,232) click 2: (310,413) approximate distance: 206  click 1: (330,392) click 2: (493,166) approximate distance: 278  click 1: (493,166) click 2: (306,173) approximate distance: 187  click 1: (293,175) click 2: (8,6) approximate distance: 331 |

All tests tiled the screen with triangles in a similar way to h1q2, except it fits as many triangles as possible in the screen region with a width of the approximate distance calculated.

Since it was not specified what to do on a 3rd click, we made it so a 3rd click would start over to allow more convenient testing/usage.

H2q2

Directory: export/home/cs450/cs450124/Hwork2

Run with: java h2q2

**Derivation**:

**Coordinate Translation Method:**

The program needs to translate coordinates in logical (x, y) units to pixels and vice versa. We desire a general solution to this problem based on the range of logical (x, y) units in a viewport. They can be represented as functions:

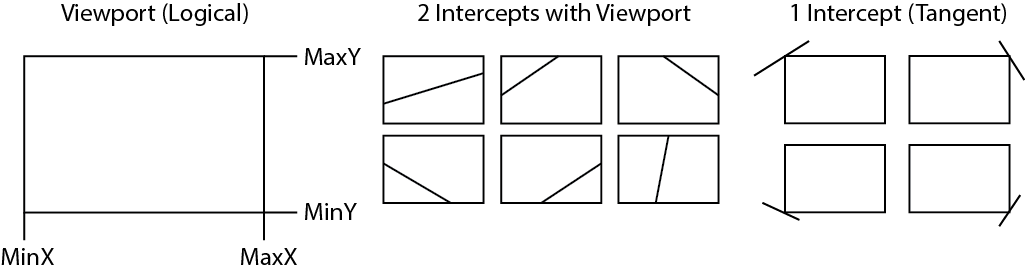
There are a number of given variables based on the fact that we are within a predefined logical space and the pixel counts are determined at runtime.

Since we have an isotropic mapping, we can determine the maximum pixels per unit because the side with extra pixels will be the limiting factor in our scaling. Using this general solution is how our program scales as the user expands and shrinks the window. We also define a drawing area of pixels to fit with the aspect ratio of the logical space.

Using dimensional analysis, we can now define our translation functions. In the pixel to logical direction, we get the number of pixels, translate it to units, and add the start of the displayed range. The opposite direction is essentially the inverse.

**Line Drawing Algorithm**:

We wish to represent the line in slope intercept form.



A line can pass through space with the viewport in 3 ways.

1. 2 Intercepts: It enters the viewport and exits.
2. 1 Intercept: It is tangent to the viewport in one of 4 corners.
3. 0 Intercepts: It does not enter the viewport.

A viewport can be viewed as the intersection of four lines and the area contained within it.

A line (in slope intercept form) must intercept these four lines unless it is horizontal. We can find this interception as follows.

If the interception points are within the minimum and maximum range of the viewport, we know that it intercepts the viewport at that point.

If there are two intercepts, we can simply interpolate the points to pixels and draw them.

**Testing**:

The program can be divided into two main functional units. The calculation of a lines interception with a viewport and the interpolation of pixels to logical units and vice versa. The interpolation functions can be checked for numeric correctness by various positive and negative values. This was already verified in a former test suite while the clicking function returns reasonable results by centering the display correctly.

The line interception calculation can be divided into a number of cases. Shifting the image should result in the same interpretation geometrically for each case. The fact that there are two lines is irrelevant because the drawing algorithm only operates on one.

|  |  |  |  |
| --- | --- | --- | --- |
| Scenario | Input | Expected Output | Status |
| Line has + Slope and + Intercept | m = 1, b = 2 | Line upward, x=0 above x-axis | Passed |
| Line has + Slope and 0 Intercept | M = 3, b = 0 | Line upward, crosses origin | Passed |
| Line has + Slope and - Intercept | m = 2, b = -4 | Line upward, x=0 below x-axis | Passed |
| Line has 0 Slope and + Intercept | m = 0, b = 2 | Horizontal line above x-axis | Passed |
| Line has 0 Slope and 0 Intercept | m = 0, b = 0 | Coincides with x-axis (nothing) | Passed |
| Line has 0 Slope and - Intercept | m = 0, b = -3 | Horizontal line below x-axis | Passed |
| Line has - Slope and + intercept | m = -2, b = 3 | Line downward, x=0 above x-axis | Passed |
| Line has - Slope and 0 Intercept | m = -1, b = 0 | Line downward, crosses origin | Passed |
| Line has - Slope and - intercept | m = -3, b = -1 | Line downward, x=0 below x-axis | Passed |