jkubath/TouchscreenInterface

github.com/jkubath/TouchscreenInterface/blob/master/spikes/Touchscreen Calibration/Coordinate Transformation Matrix.txt
jkubath

When a device (computer) is connected to the touch screen, there is a property of the

Operating System that allows the touch screen to be rotated to show the correct

orientation. This property is the Coordinate Transformation Matrix. The issue

is that this matrix can become "Reset" to the identity matrix

1 0 0

0 1 0

0 0 1

The screen will appear to be rotated correctly, but the touch will not be. This

will cause the touch capability to malfunction. Setting this matrix to the values:

0 1 0 -1 0 1 0 0 1 will calibrate the touch to the default up and down orientation

of the kiosk touch screen.

To set the Coordinate Transformation Matrix

in Ubuntu run this command:

~$ xinput set-prop "pointer:E&T INC. E&T IR SCREEN" "Coordinate Transformation Matrix" 0 1 0 -1 0 1 0 0 1

An alternative to running the command is

to run the set_xinput_matrix.sh shell script.

Tip: The shell script needs execution capabilities.

This can be done with

~$ chmod 777 set_xinput_matrix.sh

Here's a few links that helped me understand the Coordinate Transformation Matrix:

https://unix.stackexchange.com/questions/138168/matrix-structure-for-screen-rotation?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

https://wiki.ubuntu.com/X/InputCoordinateTransformation

How to calculate the Coordinate Transformation Matrix

[x] [a b c] [s]

[y] * [d e f] = [t]

[1] [0 0 1] [1]

a = your_x vs. screen_x

b = your_y vs. screen_x

d = your_x vs. screen_y

e = your_y vs. screen_y

x = your_x

y = your_y

s = screen_x = 1

t = screen_y = 1

(x * a) + (y * b) + (1 * c) = s

(x * d) + (y * e) + (1 * f) = t

(x * 0) + (y * 0) + (1 * 1) = 1

NOTE: The "(0, 0)" part of the screen is considered the upper left of the screen, so x increases from left

to right, and y increases from top to bottom. As a result, your_x is considered going from left to

right on the physical screen (regardless of what way the virtual screen is rotated), and your_y from

top to bottom. However, screen_x is considered going from left to right across the virtual screen

(which, for example, would be considered going from top to bottom on the physical screen if the virtual screen was rotated 90 degrees clockwise), and screen_y is considerded going from top to bottom across

the virtual screen. Also, when we refer to a 90 Degree Clockwise or Counter-clockwise Rotation, we're

referring to a rotation of the physical screen (and the matrix is required as a result of the rotation

of the virtual screen).

Normal Rotation

Your x is the same as the screen x and your y is the same as the screen y, so:

your_x = screen_x = 1, so x = 1

your_y = screen_y = 1, so y = 1

a = your_x vs. screen_x = 1 (because screen_x increases when your_x increases)

b = your_y vs. screen_x = 0 (because screen_x does not change when your_y increases)

d = your_x vs. screen_y = 0 (because screen_y does not change when your_x increases)

e = your_y vs. screen_y = 1 (because screen_y increases when your_y increases)

This results in:

[1] [1 0 c] [1]

[1] * [0 1 f] = [1]

[1] [0 0 1] [1]

Which means that:

(1 * 1) + (1 * 0) + (1 * c) = 1 -> 1 + 0 + c = 1 -> c = 0

(1 * 0) + (1 * 1) + (1 * f) = 1 -> 0 + 1 + f = 1 -> f = 0

So the final matrix is:

[1 0 0]

[0 1 0]

[0 0 1]

90 Degree Clockwise Rotation

Your x is opposite of the screen y and your y is the same as the screen x, so:

1 - your_x = screen_y = 1, so x = 0

your_y = screen_x = 1, so y = 1

a = your_x vs. screen_x = 0 (because screen_x does not change when your_x increases)

b = your_y vs. screen_x = 1 (because screen_x increases when your_y increases)

d = your_x vs. screen_y = -1 (because screen_y decreases when your_x increases)

e = your_y vs. screen_y = 0 (because screen_y does not change when your_y increases)

This results in:

[0] [0 1 c] [1]

[1] * [-1 0 f] = [1]

[1] [0 0 1] [1]

Which means that:

(0 * 0) + (1 * 1) + (1 * c) = 1 -> 0 + 1 + c = 1 -> c = 0

(0 * -1) + (1 * 0) + (1 * f) = 1 -> 0 + 0 + f = 1 -> f = 1

So the final matrix is:

[0 1 0]

[-1 0 1]

[0 0 1]

180 Degree Rotation

Your x is opposite of the screen x and your y is opposite of the screen y, so:

1 - your_x = screen_x = 1, so x = 0

1 - your_y = screen_y = 1, so y = 0

a = your_x vs. screen_x = -1 (because screen_x decreases when your_x increases)

b = your_y vs. screen_x = 0 (because screen_x does not change when your_y increases)

d = your_x vs. screen_y = 0 (because screen_y does not change when your_x increases)

e = your_y vs. screen_y = -1 (because screen_y decreases when your_y increases)

This results in:

[0] [-1 0 c] [1]

[0] * [0 -1 f] = [1]

[1] [0 0 1] [1]

Which means that:

(0 * -1) + (0 * 0) + (1 * c) = 1 -> 0 + 0 + c = 1 -> c = 1

(0 * 0) + (0 * -1) + (1 * f) = 1 -> 0 + 0 + f = 1 -> f = 1

So the final matrix is:

[-1 0 1]

[0 -1 1]

[0 0 1]

90 Degree Counter-clockwise Rotation

Your x is the same as the screen y and your y is opposite of the screen x, so:

your_x = screen_y = 1, so x = 1

1 - your_y = screen_x = 1, so y = 0

a = your_x vs. screen_x = 0 (because screen_x does not change when your_x increases)

b = your_y vs. screen_x = -1 (because screen_x decreases when your_y increases)

d = your_x vs. screen_y = 1 (because screen_y increases when your_x increases)

e = your_y vs. screen_y = 0 (because screen_y does not change when your_y increases)

This results in:

[1] [0 -1 c] [1]

[0] * [1 0 f] = [1]

[1] [0 0 1] [1]

Which means that:

(1 * 0) + (0 * -1) + (1 * c) = 1 -> 0 + 0 + c = 1 -> c = 1

(1 * 1) + (0 * 0) + (1 * f) = 1 -> 1 + 0 + f = 1 -> f = 0

So the final matrix is:

[0 -1 1]

[1 0 0]

[0 0 1]