Movement-RS feature guide

Linus Tibert

August 2, 2025

Contents

1 Introduction **3**

2 Recording shapes **3**

3 Shape detection **3**
3.1 Straight Lines 3
3.2 Circles 4
3.3 Ellipses 4

4 Code components **5**
4.1 Constants 5
4.2 Structs 5
4.3 Functions 6
4.4 Tests 6

1 Introduction

This paper means to be an understandable explanation of what Movement-RS does behind the scenes to record what the cursor does and to distinguish between a number of different shapes. While it is not strictly necessary to look at the code while reading this, it is highly recommended for understanding the text and the code itself.

Note that hyphenation is enabled for variable names too. This may get a little confusing, but the variables in Rust never use hyphens, so it should still be clear what the name of a mentioned variable is.

2 Recording shapes

In this section, I will explain, how new shapes are recorded in the code.

First, a new instance of the `Recording` struct is created. The `update()` function of the `Recording` is executed periodically with an interval of $\frac{1000}{\texttt{FRAMERATE\_FPS}}$ milliseconds. When the function is executed for the first time, the `init()` function is called to prepare for a new recording. As soon as the cursor coordinate changes, the recording starts and each `update()`, or 'frame', the current cursor coordinate is added to the `coordinates` vector. Each frame, the `update()` function returns a `RecordingStatus` which tells the main function the current status.

If the coordinate does not change, 1 is added to the `coordinate_unchanged_cycles` property (Starting at 0 after initailization). If this property is equal to `END_FIGURE_TIMEOUT`, the `update()` function returns a `RecordingStatus::Finished`, telling the main function to reset the recording after moving the `coordinates` into a new `Shape` object which is pushed into the `shape_collection`.

As soon as a new `Shape` is added to this collection, the program iterates over all objects in it and runs the `Shape.get_shape_name()` for those which have their `shape_type` set to `ShapeName::Undefined`.

3 Shape detection

3.1 Straight Lines

Line detection in this program is done in a very straight forward way. First, the program checks, if the first and the last point recorded have the largest distance of all points in the recording. These points are represented as A and B on figure 1. Next, the program creates a vector $\overrightarrow{AB}$ and checks if all other points (in this example C_1 and C_2) are in a certain distance from the vector. This distance is defined by the constant `LINE_TOLERANCE_PX`. The last step is checking if the rate of points which are too far away is less than or equal to `TOLERANCE_GENERAL`, which is 0.25 by default.

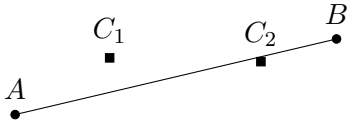

Figure 1: Line detection

Assuming that A, B and C_2 in this example are in the tolerated distance, figure 1 would be detected as a line because 0.75 of all points have passed the distance check.

3.2 Circles

The circle detection is much like the line detection. But it does not check if other points are in a certain distance of to a vector. Instead, the centre of all points in determined (the average position of all points) and the distances of all points are compared with the average distance a point has to this centre. The tolerated deviation of the average is defined by `CIRCLE_TOLERANCE`. Once again, the resulting rate of passed points needs to be equal to or less than the `TOLERANCE_GENERAL` for the shape to be detected as a circle.

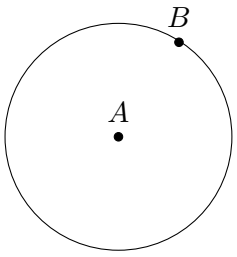

Figure 2: Line detection

On figure 2, A represents the centre of the shape, while B in a point with a distance close to the average distance, which is represented by the circle.

3.3 Ellipses

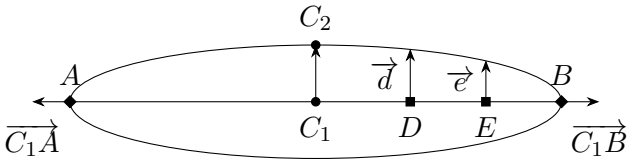

Figure 3: Ellipse detection

Ellipse detection works in a kind of weïrd way and was definitely the hardest to implement. It is divided into two main aspects: the factor of growth and the symmetry of the figure.

To prepare for determination of those factors, the first step is to get the two points which are the furthest away from each other; in figure 3, these are represented by A and B. After this, the centre between those point is defined (C_1). The location of this point can be compared with the average coordinate; this is done with a tolerance of `ELLIPSE_CENTRUM_TOLERANCE_PX`. If the centre of the vector $\overrightarrow{AB}$ isn't inside the tolerance area, the shape is discarded as not an ellipse.

Factor of growth

Now, the script measures the length between C_1 and the point closest to it (C_2) and stores this length for the next step, in which two vectors are created: $\overrightarrow{C_1A}$ and $\overrightarrow{C_1B}$. The program divides those vectors into a number of points (half as many as the whole figure has for each vector) along them (D and E). Using those points, the program can now move along vector $\overrightarrow{C_1A}$ and $\overrightarrow{C_1B}$ to check at each stop, if the

distance to the closest point of the figure is shorter than the distance measured at the previous stop (I.e., e.g., if $\overrightarrow{d}$ is shorter than $\overrightarrow{e}$.).

Symmetry

At each stop, the generated vector (e.g. $\overrightarrow{d}$) is mirrored to check if there is another point with a similar distance on the other side of $\overrightarrow{AB}$. How far such a point may be is determined by the `ELLIPSE_TOLERANCE` constant.

Evaluation

To evaluate the two factors above, a level of `perfection` (p), is calculated using the factor of growth (F_g) and the factor of points without a mirrored twin (F_t):

$$p = \frac{(F_g + F_t)}{2}$$

If the perfection is bigger than `TOLERANCE_GENERAL`, the shape is seen as an ellipse. While this method is working well to distinguish circles from ellipses, it is not able to distinguish an ellipse from a rhombus. Another problem is, that a lot of tolerance is needed to be able to detect a hand-drawn ellipse.

4 Code components

This section gives a brief overview of some core parts of the code. This is not meant to be a proper documentation of the code but just a quick summary of some functionality.

4.1 Constants

N.b. most of the constants are explained in section 2 and 3 in more detail.

`END_FIGURE_TIMEOUT` is the amount of 'frames' which have to be the same in a row for a running recording of mouse movement to get stopped and evaluated.

`FRAMERATE_FPS` the amount of 'frames' taken every second while recording. Every frame is a `Vector2D` added to the `coordinates` field of the `Recording` vector.

`TOLERANCE_GENERAL`, `CIRCLE_TOLERANCE` and `ELLIPSE_TOLERANCE` are factors used as tolerances while checking if a `Recording` is of a specific `Shape`.

`LINE_TOLERANCE_PX` and `ELLIPSE_CENTRUM_TOLERANCE_PX` are pixel amounts used as tolerances while checking if a `Recording` is of a specific `Shape`.

4.2 Structs

The `Recording` struct has a more detailed description is section 2.

`Vector2D` is a two-dimensional vector which is also used as coordinate in the script. I didn't use any existing class to be able to implement all functions myself.

`Recording` is a struct used to manage the current recording-session of mouse movement. As soon as it is finished, the coordinates will be cloned into a new `Shape` and the `Recording` object will be discarded.

`Shape` is used to store a set of coordinates and the associated `ShapeName`. The struct mainly exists because originally, the `shape_collection` Vec should be exportable for other programs. Maybe I'll add this feature in the future.

`DistanceSet` is used to store the shortest and longest possible vectors in the `Shape` and the associated lengths.

`PointDistanceSet` is much similar to `DistanceSet`. It is returned by a single function getting the minimum and maximum vectors between a point and all other points in the `Shape`. The function also adds a some evaluation data relevant for circles.

4.3 Functions

I will only mention functions here, which don't have such a self explaining name.

`Recording.update()` runs in a loop every frame to manage the recording.

`Vector2D.abs()` returns the length of the vector.

`Shape.get_shape_name()` is the function containing the shape determination functionality.

`Shape.find_centre()` returns the average `Vector2D` coordinate of the whole shape.

`Shape.get_distances()` returns the `DistanceSet` mentioned in section 4.2. The explanation there should enough to understand this functions purpose.

`Shape.get_point_distances()` returns the minimum and maximum distance vectors and associated distances of a given point, `passes_percent` which determines the probability of which the `Shape` is a circle, and a bunch of other data relevant for circles.

4.4 Tests

There are four tests to confirm that the code is really able to determine certain shapes. All of them use hand recorded `Shapes`, which passed the checks pretty good, but no recording contains a perfect shape to test the code. I will probably add perfect shapes to the tests in the future.

The `test_junk()` test has a number of recordings in it, which should not be recognized as any of the currently available shapes. Some of them may be correctly recognized if a new shape is added to the collection of known shapes.