UNISTROKE GESTURE RECOGNITION
Documentation

Contacts

Telegram: @NKonovaloff

Email: nd.konovaloff@gmail.com

If you find any mistakes in the text, please contact me

UNISTROKE GESTURES:

Unistroke gestures are a way to input by drawing various lines. It can be simple straight lines, broken and curved or closed into shapes. This package contains tools for working with Unistroke gestures and their recognition.

ABOUT:

This package contains tools for working with Unistroke gestures:

- The editor for creating and editing a gesture pattern path and the inspector prefabs.
- Gestures are considered insensitive to position and size, sensitive to direction, and can also be defined as sensitive or insensitive to aspect ratio.
- The recognition method is based on the $1 algorithm but modified to support aspect both ratio sensitive and insensitive gestures. The recognition method was developed using the Unity Job System and the Burst compiler to provide the best performance and minimize managed memory allocation.

GESTURE PATTERNS:

The gesture recognition algorithm is based on comparing the input gesture with a set of patterns. As a result of recognition, the most similar pattern is selected.

The package contains an abstract class ***GesturePatternBase*** for creating your pattern type. **GesturePatternBase** is derived from ScriptableObject so that you can easily manage and edit pattern prefabs using the ***Pattern Editor***.

The base class defines the basic information necessary for the recognition method to work - the gesture path. The path of a gesture is what the gesture itself defines. The gesture path is represented as a set of points. If you connect all adjacent points along a line, a gesture will be displayed. The first point corresponds to the beginning of the gesture path, and the last point to its end. When recognizing, paths are compared in this order, from the first point to the last, which makes gestures sensitive to direction.

Inside the gesture prefab, the points are stored as an array of Vector2 points. To get a set of points, refer to the ***Path*** property, which returns points as a ReadOnlySpan<Vector2>. The gesture path is represented in normalized form, which means that all points have coordinates inside a square with unit length sides (from 0 to 1).

The base class also contains a ***ScalingMode*** field that defines the gesture type:

- The ***Uniform*** value makes the template aspect ratio sensitive. This is useful if you need to be able to distinguish, for example, squares from rectangles, since when processing the path, the original ratio of one hundred gesture paths will be preserved. This value is also required to indicate 1D gestures (strictly vertical and horizontal lines).
- The ***UnUniform*** value makes the gesture insensitive to aspect ratio. This causes, for example, when processing a path representing a rectangle, the result will be a square. This option helps to reduce the input variance of the gesture path, which increases the accuracy of gesture recognition.

You can open the examples folder and find the pattern prefabs used for recognition ("UnistrokeGestureRecognition/Example/Prefabs/Gesture Patterns") and try changing this field to see how the path preview changes in the inspector. This field does not affect the display of the path inside the template path editor field.

To create your own patter type and add your own data, inherit your class from this type. Your class will be supported by the template editor and custom inspector.

```
using UnistrokeGestureRecognition;

// Do not forget to add CreateAssetMenu attribute
class MyCoolPatternType : GesturePatternBase {
    // My useful data
}
```

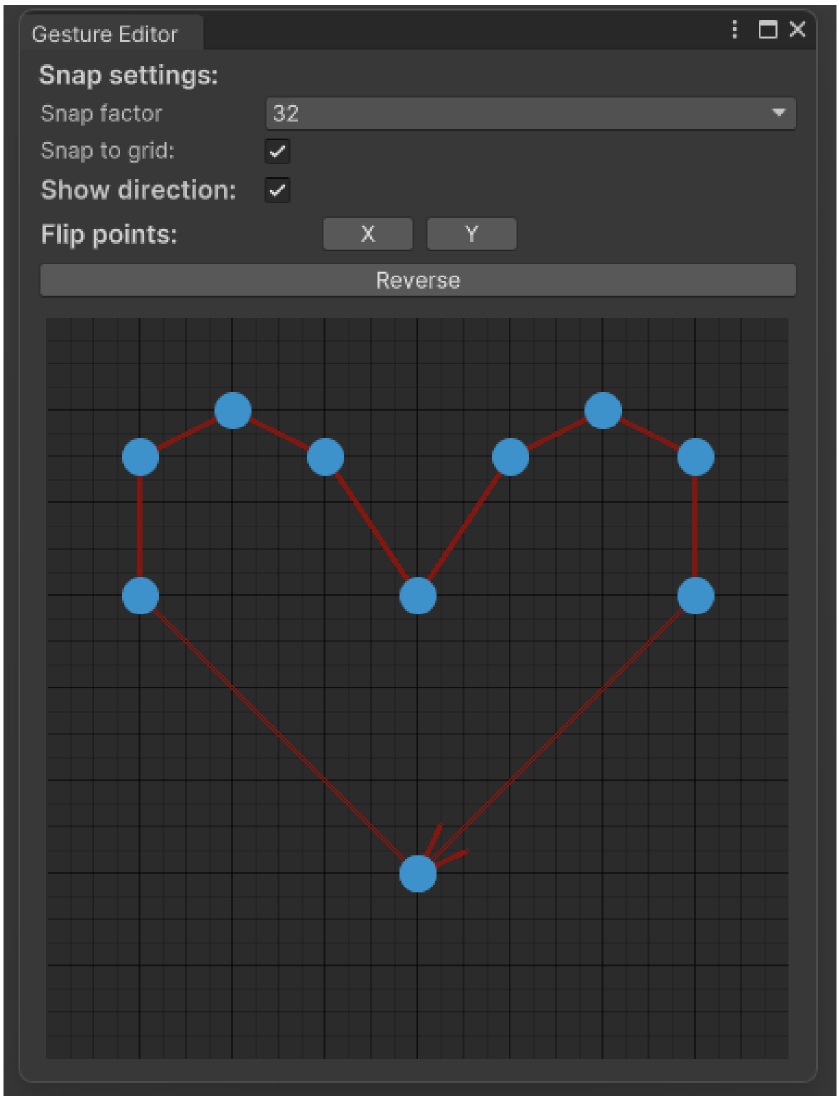
PATTERN EDITOR:

The ***Pattern Editor*** is designed for creating and editing the path of a gesture template. The editor window can be opened from the top menu in the "Tools > Gesture Editor" tab, or by clicking the "Open in Editor" button in the inspector window with a template prefab selected. The editor window will display the path of the currently selected template or will be inactive if no template is selected.

The editor has several settings: snapping waypoints to a grid of different levels and displaying the direction of the gesture in the form of an arrow at the end of the path.

Gesture path editing happens on the grid canvas at the bottom of the window. Gesture path editing happens on the grid canvas at the bottom of the window. The editor has two modes of operation: the mode of adding points (***Normal Mode***) and the mode of cutting lines (***Cut Mode***).

Normal Mode is for adding, removing, and moving points on the canvas. When normal mode is active, hovering over the hole displays the position of the added point on the canvas and displays the path from the last point in the path, if any. When you move the cursor over the set point, it is marked with a brighter color and is considered selected.



Gesture Editor

Snap settings:

Snap factor: 32

Snap to grid: ✓

Show direction: ✓

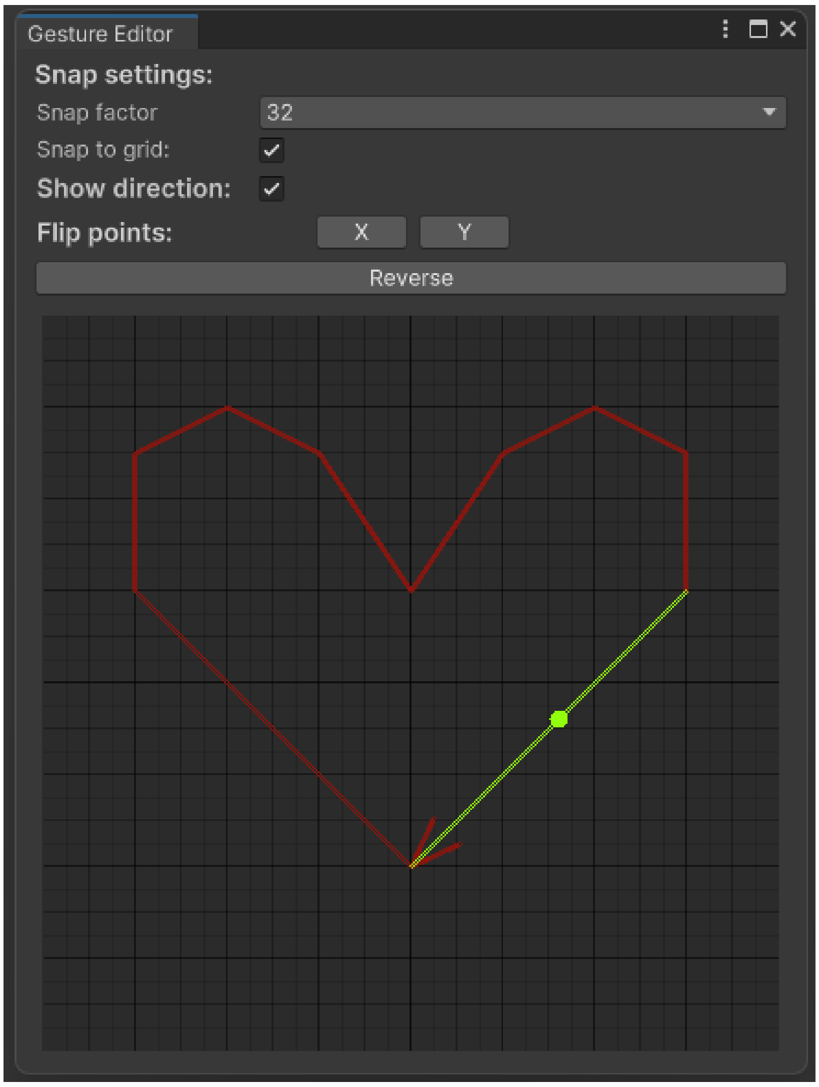
Flip points:

*Editor with open pattern in **Normal Mode***

Controls in the editor in ***Normal Mode***:

- *LMB* - Add a new point to an empty space in the hostel at the cursor location. When adding a new point and holding LMB, allows you to move the added point on the canvas.
- *LMB (Hold)* - Allows you to move the selected point on the canvas with the mouse.
- *RMB* - Delete selected point.
- *MMB* - Switch to ***Cut Mode***.

Cut Mode is designed to add new points between previously set points. In ***Cut Mode***, only lines are displayed, points are hidden. Also, the prompt for adding a new point is not displayed. When you move the cursor over a line, this line is highlighted, and a marker is displayed at the cursor location, showing the place where the new point will be added.



Snap settings:

Snap factor 32

Snap to grid: ✓

Show direction: ✓

Flip points: X Y

Reverse

Editor with open pattern in ***Cut Mode***

Controls in the editor in ***Cut Mode***:

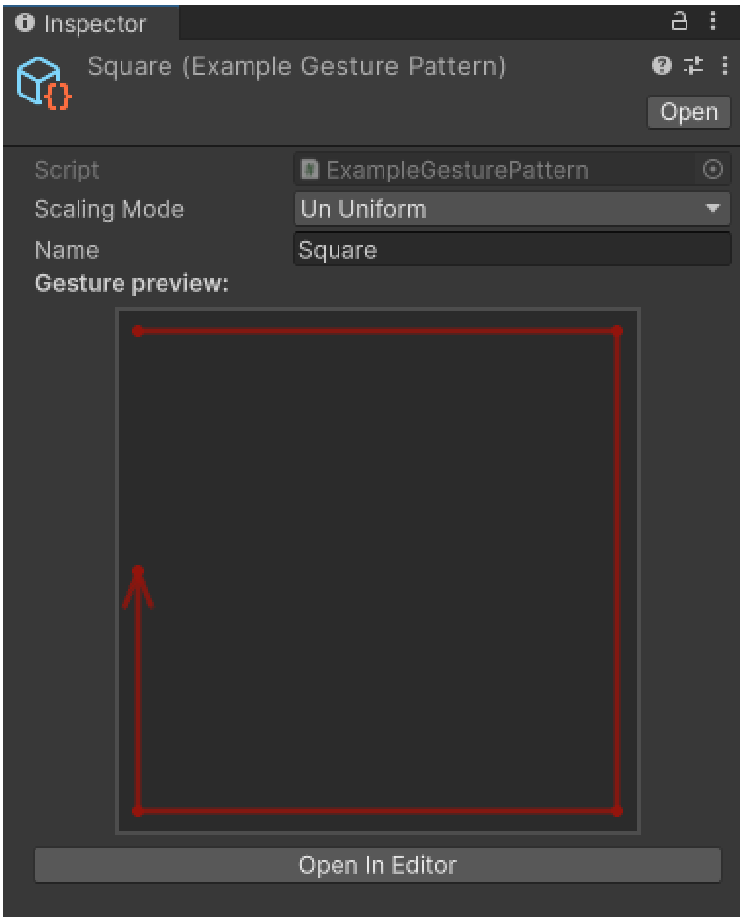
– *LMB* - Add a new point at the cut marker location. When adding a new point and holding LMB, allows you to move the added point on the canvas.
– *MMB* - Switch to ***Normal Mode***.

You can also use *CTR+Z* to undo the last action and *CTR+Y* to redo the action in both modes.

The "X" and "Y" buttons in the editor allow you to reflect the entered path in x and y coordinates, respectively. The "Reverse" Full button invents the entered path so that the start and end of the path are reversed.

PATTERN INSPECTOR:

At its core, the ***Pattern Inspector*** is identical to the standard ScriptableObject prefab inspector. Except that the ***Pattern Inspector*** shows a preview of the path of the selected pattern.



Script: ExampleGesturePattern
Scaling Mode: Un Uniform
Name: Square
Gesture preview:

Pattern Inspector *example*

Note that the path in the inspector may not match the path in the editor. The path in the editor is presented in raw form. When displaying the path in the inspector, the set of points from the prefab ***Path*** field is used. Changing the path in the editor is immediately reflected in the path in the preview.

RECORDING THE GESTURE PATH:

To recognize a gesture, you first need to record the path that the user enters. This package does not contain a module for the gesture recording itself. Just a way to optimally store the waypoints while recording it.

This is so because the recording method can differ significantly from the game. Someone might want to read the path from the cursor position on the screen, and someone might want to record the path drawn on the game object. So, you need to implement your own write logic. You can explore a simple example of how to implement a gesture recording in a ***Recognizer Example Scene***.

To simplify the implementation of recording, the package contains a ***GestureRecorder*** class for storing recorded points. This class allows you to write gestures of any length to a limited buffer. This avoids unnecessary allocation of managed memory and improves performance. In its implementation, the class also uses a native buffer to store points. So, it's important to remember to call ***Dispose*** method when the recorder object is no longer needed.

To create ***GestureRecorder*** object, call the class constructor with two parameters:

```
var gestureRecorder = new GestureRecorder(254, 0.5f);
```

The first parameter specifies the maximum buffer size for storing points. When recording points, when the number of recorded points reaches the specified value, the recorded path will be resampled in such a way that the overall shape of the path is preserved, but the number of points is halved, thus freeing up space for recording new points. It is worth considering that the resampling process is not perfect - with each new execution, the recorded path is slightly distorted. *And the more often resampling is performed, the more the recorded path differs from the actual one*. Thus, it is worth choosing a buffer size large enough so that resampling is performed less frequently, while the size should not be too large so as not to take up too much memory. It is also worth noting that the ***GestureRecorder*** implementation allocates not one, but two buffers of the specified size to be able to transfer values between them during resampling. A good value would be a capacity of 500 points, but for your game the best value may be your own.

The second value allows you to set the minimum distance between the last recorded point and the added one, necessary for recording this point. This allows you to reduce the number of points recorded at high fps, as well as avoid recording points when the user continues to enter the gesture but does not move the cursor.

You can see how the ***GestureRecorder*** works in the ***Recorder Example Scene***. In this example, when entering a gesture, the actual path is marked in red, and the path recorded using the ***GestureRecorder*** is marked with blue dots. Notice how the points are repositioned when the maximum buffer size is reached.

RECOGNIZING THE GESTURE:

The package contains 2 gesture recognizer implementations. The ***SpecificGestureRecognizer*** is designed to recognize gestures of the same type: either only aspect ratio sensitive gestures, or only insensitive ones. This implementation must be chosen if you have certain templates with the same types of gestures for recognition. If your template set has both types of gestures, then you should use the ***GestureRecognizer***.

Use the following constructor to create a recognizer object:

```
var recognizer = new GestureRecognizer<ExampleGesturePattern>(patterns, 128);
```

The first parameter is your set of recognition patterns.

The second parameter is the number of points to which the pattern and gesture paths will be given when recognizing. The higher this value, the more accurate the recognition result will be, but the execution time will be longer. Depending on your set of patterns, the optimal value of this parameter will vary, but in most cases 128 points gives a satisfactory result.

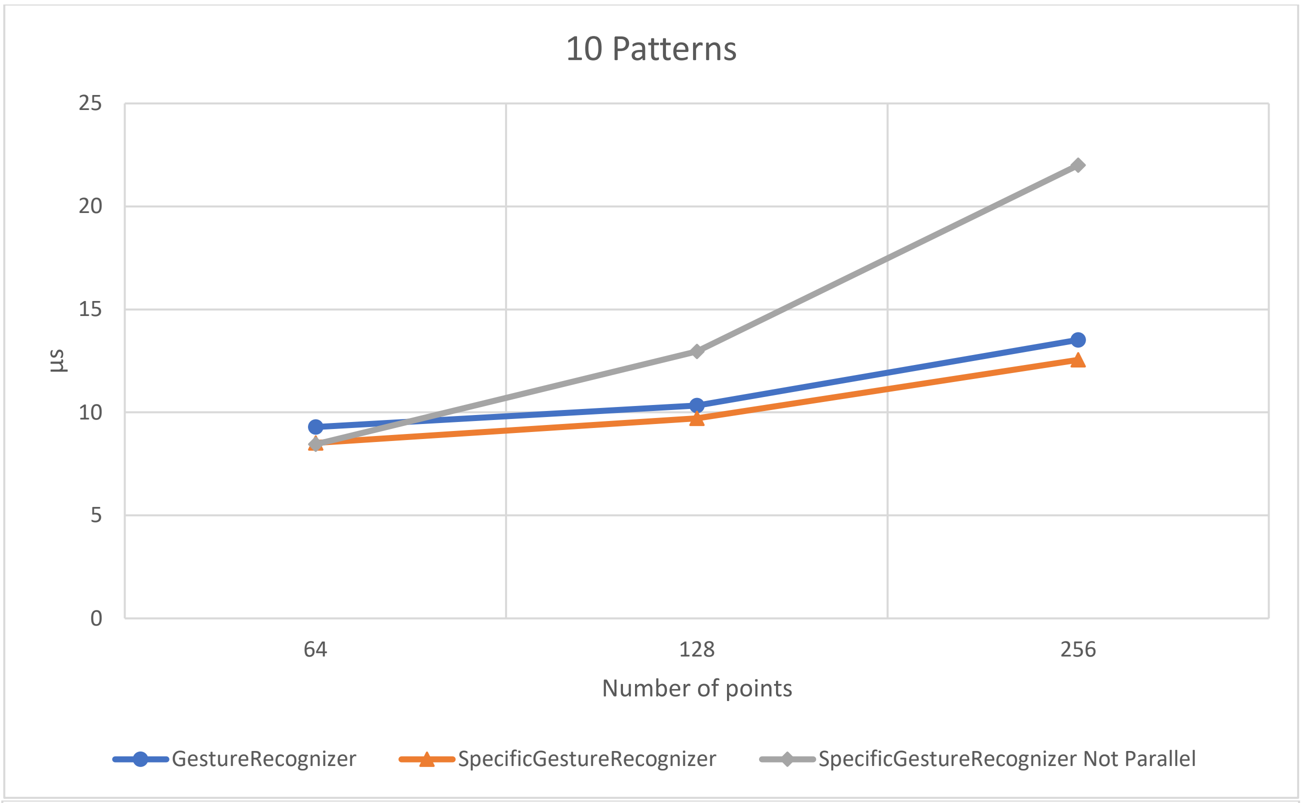
SpecificGestureRecognizer also takes a third parameter that specifies the type of gesture to recognize**.**

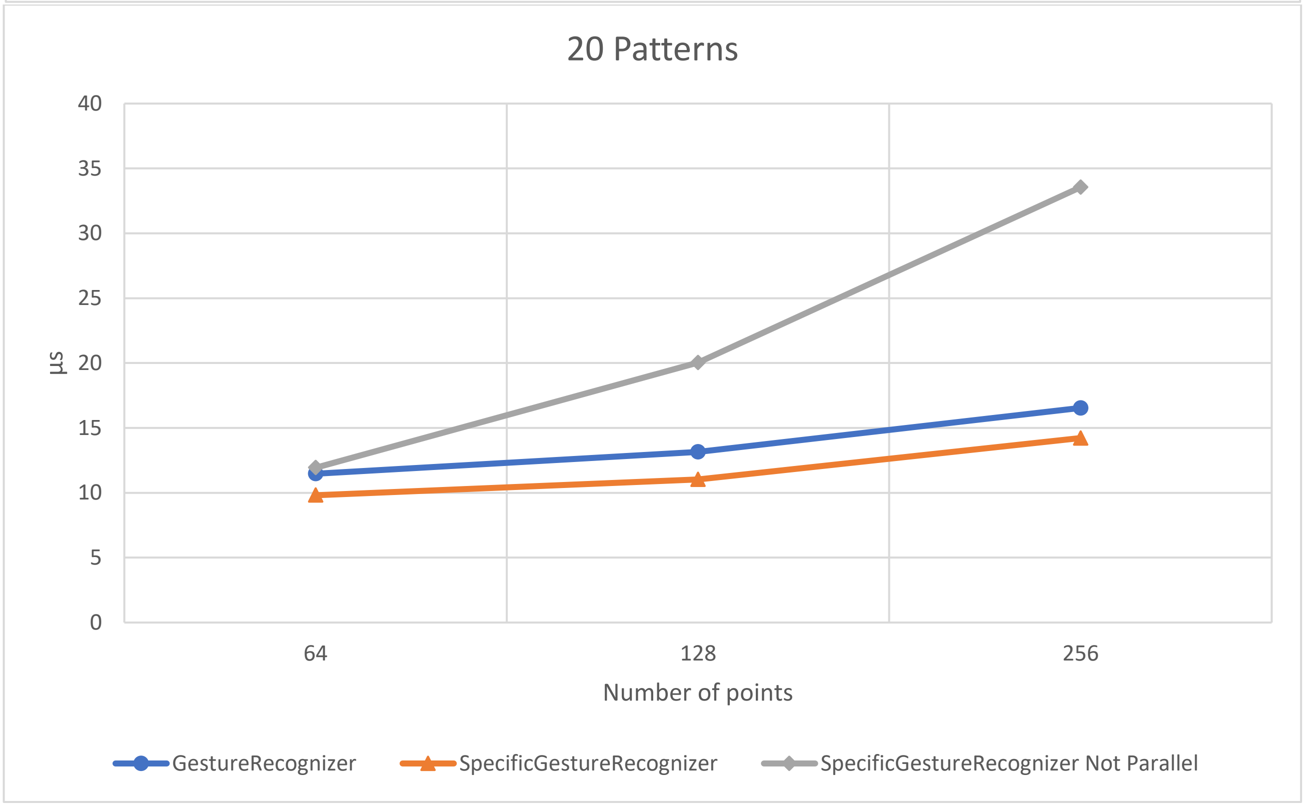
Both implementations use Unity Job System and Burst Compiler, so the runtime of the recognition method is efficient enough that you don't have to worry about performance. Calling the recognizer method also does not cause managed memory allocation. Here are the results of performance tests of recognition methods with different parameters.

Tests configuration:

CPU: Intel® Core™ i5-10500H @ 2.50GHz

RAM: SODIMM Crucial Ballistix 8GB * 2 DDR4





After recording the gesture, you can get the final path through the ***Path*** field. Pass this value as a parameter to one of the recognition methods: ***ScheduleRecognition*** or ***Recognize***. You can also specify whether the pattern matching of the recorded path should be performed in parallel or sequentially. Tests have shown that non-parallel comparison can be more efficient if the number of patterns and the number of points is small enough.

The ***Recognize*** method blocks the main thread until the end of the recognition method execution and immediately returns the result.

The ***ScheduleRecognition*** method performs scheduling of recognition tasks and returns a ***JobHandle***. You can track the completion status of recognition using this value. After the recognition is completed, you can get the result by accessing the ***Result*** field. Using this you can schedule the recognition in the ***Update*** loop and get the result later in the ***LateUpdate*** loop to ensure the least delay on the main thread.

THE RESULT:

Regardless of the method used, the result of recognition is the ***RecognizeResult*** value containing a pattern object ***Pattern*** and a value ***Score*** that determines the similarity of the recognized pattern and the path passed to recognition in the range from 0 to 1. *The higher the value, the more similar the pattern and the recorded gesture.*

One of the disadvantages of the algorithm is that the most similar pattern will always be selected as a result of recognition. Based on the value of the ***Score***, you can reject the results if it is less than a certain limit:

```
RecognizeResult<ExampleGesturePattern> result = recognizer.Result;

if (result.Score >= .7f) {
      // The pattern and gesture are similar enough
}
```

In most cases, a threshold value of 0.7 is sufficient to eliminate most of the cases where the user enters the wrong gesture. But depending on your pattern set, you may find a more appropriate value.