MTGRSuite How To

# General

Touchpad application and accompanying library are written in C#. Touchpad application utilizes [Windows Touch](https://msdn.microsoft.com/en-us/library/windows/desktop/dd371406(v=vs.85).aspx) for Windows 7. Later version of Windows might partially support Windows Touch, but identical behavior can’t be guaranteed.

## Setup

Use git to clone from <https://gitlab.com/openhid/MTGRSuite>. Open *MTGRSuite.sln* with Visual Studio 2013 or later.

Repo contains a solution with two components:

1. MTScratchpadWMTouch – The touchpad application that captures user input, serves to:
   1. Capture, record, load stored user input
   2. Create new templates
   3. Test Gesture recognition Algorithms
      1. Experiment allows structured testing by users of a recognizer over a set of gestures
      2. Batch jobs such as running a canned set of gestures
2. MTGRLibrary – class library that provides implementation of various gesture recognition algorithms and accessory classes

# MTScratchpadWMTouch

## General

The touchpad interacts with MTGRLibrary by using its Gesture Classes and Recognizer Classes. Input is stored within the PointMap and Point classes, also from the library. Points are captured via onUp, onDown, and onMove. Once enough points have been captured (the number of fingers \* CANDIDATE\_CLASSIFY\_ONCOUNT), the size of a window, runRecognition() is called.

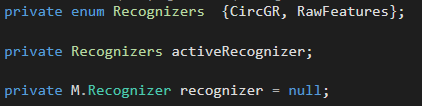
Touchpad can be in one of two main modes: Template or Candidate, which can be seen in the lower left hand corner of the touchpad window. The active template gesture set is in private List<M.Gesture> Templates and is by default, loaded from, gestureSubDirectory, which is the TemplateGestures folder by default.

## Template Mode

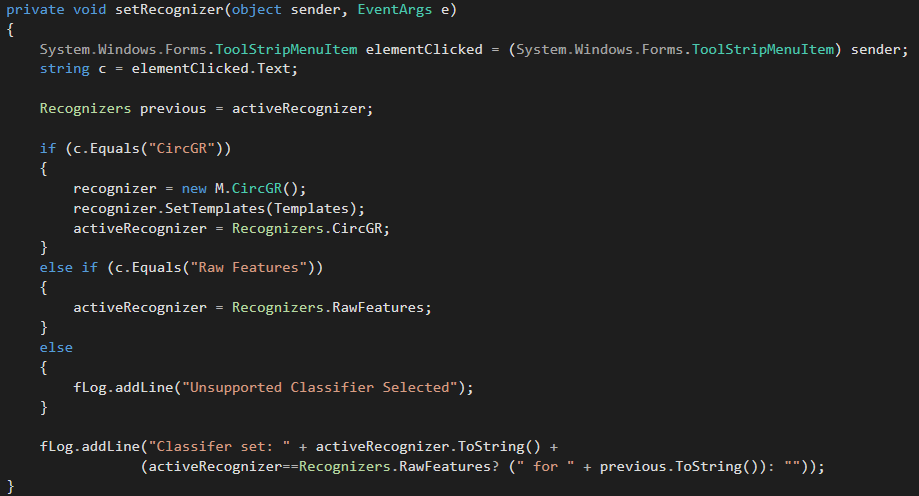
Template mode is for creating and adding new gesture to the template folder. Draw a gesture and add it via CRTL +A, or the Gestures menu. Newly added gestures **ARE NOT** part of the active template gesture set. The gesture set must be reloaded to include all new gestures with CTRL+R.

## Candidate Mode

Gesture recognition occurs in candidate mode. The recognizer is determined by a reference:



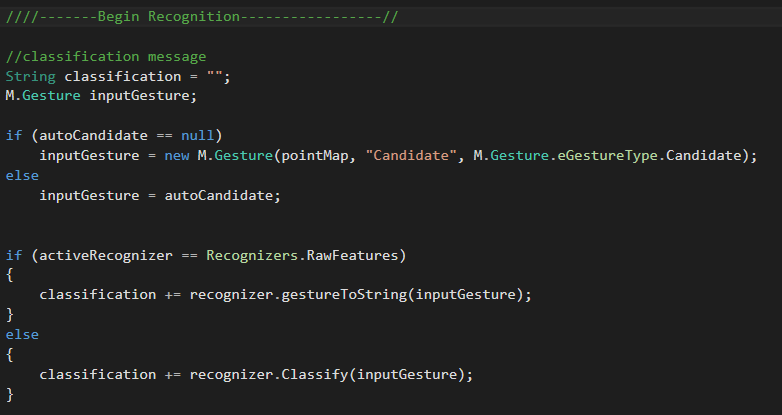
The recognizer reference is set to some class that inherits from the abstract Recognizer class in the MTGRLibrary, that will perform the classification. For convenience, touchpad keeps an enum of all the available recognizers(here CircGR and RawFeatures) as well as a reference, activeRecognizer, that points to the currently used recognizer. Recognizer can be changed by selecting a new one from the “Classifier” item in the tool strip at the bottom of the touchpad, which calls the following:



The recognizer is set to a new instance of a Recognizer subclass, and its templates must be set with the Templates stored by the touchpad. RawFeatures does not change the active classifier as its purpose is to output that classifier’s gesture representation for the input gesture to the log window instead of the classification.

## Recognition

The key part of recognition is shown below:



Batch recognition jobs work by mostly using the same recognition pipeline as normal recognition, the only difference being passing in each gesture in the batch as autocandidate. If an autocandidate isn’t passed in, gesture is created from input pointMap, and passed into the recognizer for classification.

## Misc

The menu in the touch pad also contains “Gesture”, “Test”, “Training”, and “Experiment” options, which are described below.

Gestures

* Allows adding and reloading template gestures. Load custom can allow loading gestures from other directories instead of the default. Remove Custom just reloads the defaults.
* “Run Canned” runs every gesture in the cannedCandidatesSubDirectory directoryte through the classifiers. Doesn’t do any testing itself. Can be combined with Test by clicking “Start Test”, “Run Canned”, and then “End Test.”
* “Run Shrinking Window” is a modified version of Run Canned. It takes 3 integer parameters(currently hardcoded); start, end, and step. Test is automatically invoked, all canned gestures are run at the window size specified by start, result are displayed, then the process is redone for all windows from start to end, increasing by the step variable.

Test

* Initiate a test with “Start Test.”
* Allows testing the recognizer by having the touchpad instantiate a confusion matrix, ask the user to enter a random gesture from Templates, and recording the classifier’s result before asking the user to perform another random gesture.
* Test ends when user selects “End Test,” where the results are displayed on the log window.
* Note: The difference between this and Experiment is that Experiment is a structured test, with feedback, template capturing, and specific beginning and ending in a uniform process. This is just a simple test that the user can stop at any time.

Training

* Training is similar to Test with the exception that no results are gathered. The point is to generate examples for each template.
* Touchpad asks for a gesture expects users to input that gesture. It then saves that gesture in examplesSubDirectory correctly labeled as the gesture is supposed to be for future batch runs.

Experiment

* Implements experiment process for “Evaluation of a 2D Gesture Recognition Algorithm”.
* Create an experiment by selecting a mode from “Mode”.
* Select “Start” to actually begin the experiment.
* Selecting “End” will end the experiment.
  + This will end the experiment prematurely if the experiment is still ongoing.
  + If the experiment has been completed, this will destroy the experiment object to allow another experiment to be created.
* All results of the experiment are stored in a newly created directory within experimentSubDirectory.

# MTGRLibrary

## General

Contains the implementation of gesture recognition algorithms. Currently, these are all template matching algorithms. Classes such as Geometry, Point, and PointMap provide uniform representation for Geometric primitives and operations.

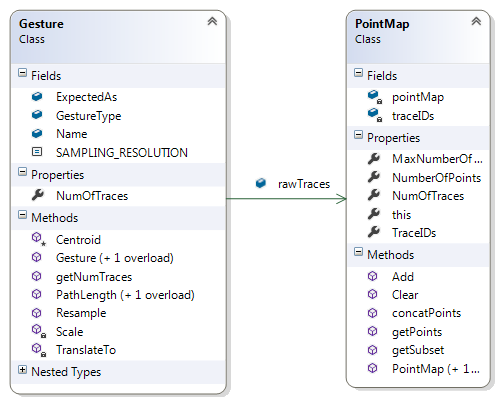
## Classes

### Recognizer

All recognizers are subclasses of Recognizer class, which provides a uniform interface for any new recognizer to establish interoperability with the TouchPad class in MTScratchpadWMTouch.

### Gesture

The Gesture class provides a generic gesture object that consist of an instance of a *Name*, an instance *PointMap*, and a *GestureType* (a nested Enum that can be “Candidate”, “Template”, or “Example”). *PointMap* class is a utility class; a hash map for traces where the key is the touch ID of trace and the value is a list of points for that trace.

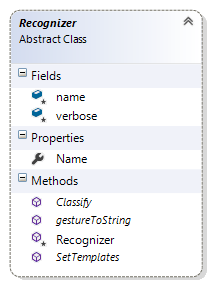


*SAMPLING*\_*RESOLUTION* is the desired resolution to use while resampling. *ExpectedAs* is an optional field that labels a gesture with its “true” label, such as when you know that this gesture is an example of “A1” template. Generic methods for scaling, translation, and resampling are provided to use if desired.

Implementing a new recognizer might require preprocessing the raw input traces into alternative observations (for example CircGR generates circular observations from the traces). This should not be included in Gesture. A new class should be created that inherits from Gesture and that utilizes the raw traces to create the relevant representation. The recognizer that uses this representations would then use this class for its classification.

## Recognizer

Recognizer class provides an interface and interop between any recognizer and TouchPad/ThirdParty Code.

The only constructor takes a *name* for the recognizer, which is stored as a field. The Boolean *verbose* is used to toggle console output.

Methods:

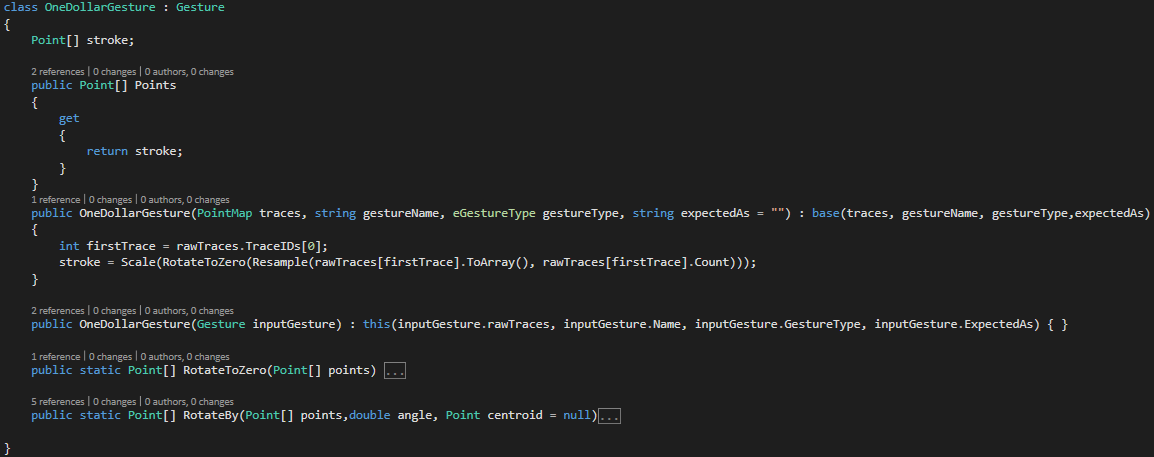
* Classify(Gesture) – returns classification for a generic gesture object. If the recognizer requires a unique representation of Gesture and any preprocessing, it would be done within this method.
* gestureToString() – returns a string version of the gesture representation used by the classifier. Useful for making sure preprocessing is being done correctly.
* SetTemplates(List<Gesture>) – a template matcher requires templates to work with. The templates might also have to be converted into the recognizer’s gesture representation and other template specific preprocessing might have to be done before the classifier can be used. This can all be done here.

## Creating a new Recognizer Example

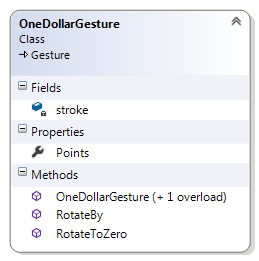
Let’s add the $1 Recognizer to the MTGRLibrary and link it to the Touchpad application. For organization, all Classes are added in their own namespace (create a folder in MTGRLibrary, and add classes there).

### Creating the Gesture Class

Gestures in $1 take the raw input trace and resample it, rotate it and scale it. We can do all this through a OneDollarGesture class that inherits from Gesture. We provide a constructor that allows conversion from generic gestures to $1 Gestures. Since $1 gestures are single trace, we decide to ignore all traces sent in except the first one.



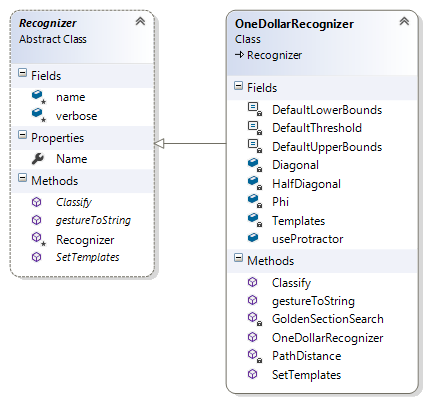
The Class Diagram:



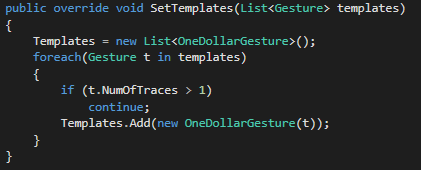
Note, this class is small enough that it could be embedded in the Recognizer class, but for more complicated Gesture representation, this gets messy.

### Creating the $1 Recognizer

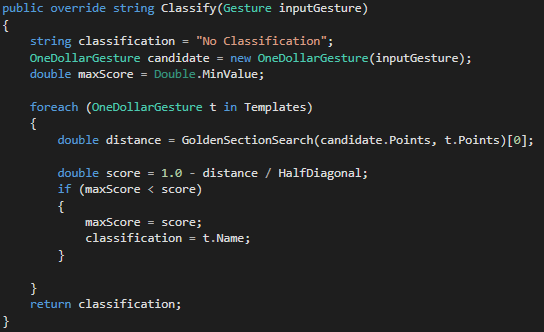
We create the classes OneDollarRecognizer and make it inherit from Recognizer, as shown below:



SetTemplates is overridden and made to store the appropriate templates from generic gestures passed in to the recognizer. Since $1 is single stroke/trace, we ignore templates with more than one trace



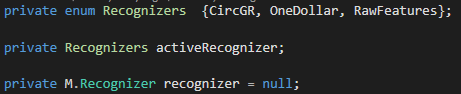
We override the Classify method do perform the actual functionality according to $1’s algorithm, involving the use of GoldenSectionSearch:



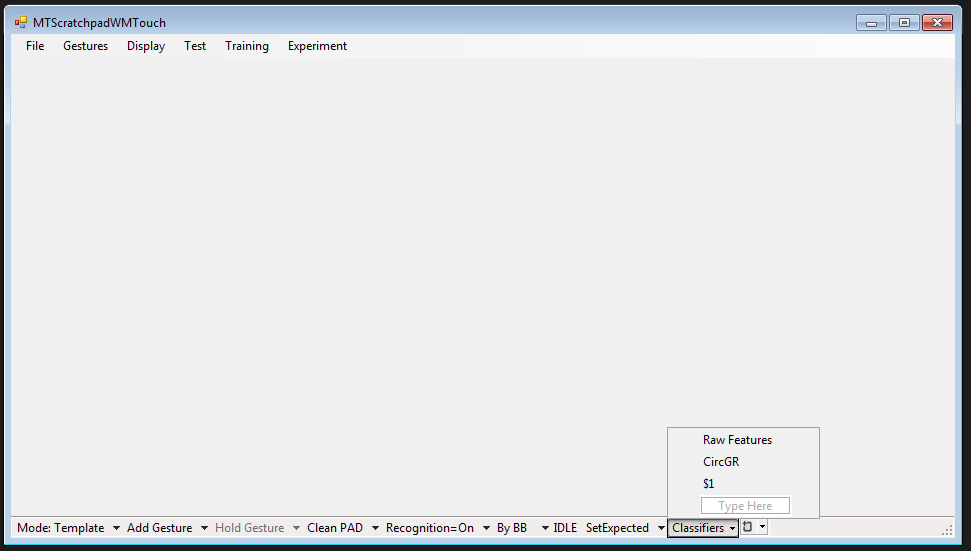
GestureToString can be overridden to return a string with the centroid coordinates, number of points, and the list of points of the gesture or some other data to verify representation of gesture is occurring correctly.

### Adding it to Touchpad

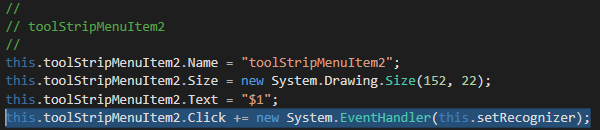
Add a new entry for $1 in the Recognizer enum.



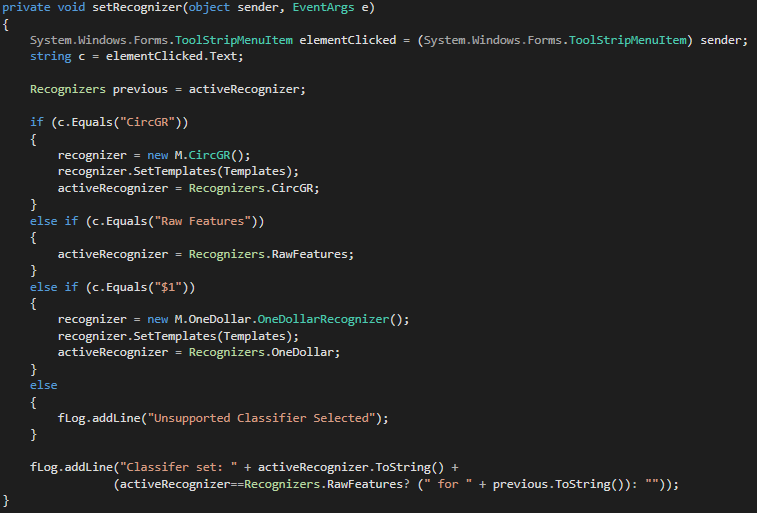
Add a new UI element to the “Classifiers” drop down menu using the Visual Studio Form Designer (shown below), this will automatically add code to TouchPadDesigner.cs for the new element.



You can find the code for the new UI element in TouchPad.Designer.cs by searching for the text you put in the UI element (in this case “$1”). Once you found the code, add an *EventHandler* for Clicks for that UI element that points to setRecognizer, as shown below:



Update the setRecognizer method to set $1 as the active recognizer.



Done!