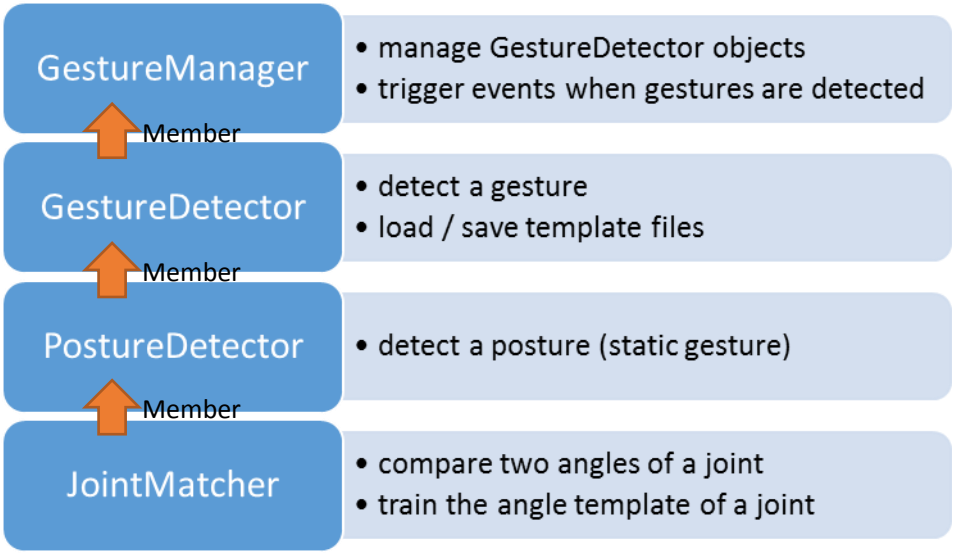
Gesture detector: Brief introduction to the algorithm

Simply speaking, the program splits a gesture into several key postures. When these postures are detected successively, then the gesture event is triggered.

The detection is mainly done by four classes. They form a hierarchical structure as below:



The main working flow is in `GestureManager.Update()`. Some details of the algorithm will be introduced below.

1. How to calculate the orientation of a joint?

We use a Vector3 object to represent the orientation of a joint, i.e. the difference of the coordinate of the next joint and the current joint. For example, the orientation of the shoulder is position[elbow] – position[shoulder].

Before calculating the position, we will also adjust the orientation of the torso to +z, so as to make the posture detections somehow robust to the torso's orientation. When the user stand facing the Kinect, the +x, +y and +z directions are his right, up and front.

These works are done in class `JointFeature`.

2. How to compare two orientations of a joint?

Generally, we want to compare the orientations of the template and the real time joint to see if they match. Basically, if the angle of the two orientations is not larger than a threshold, they will be considered matched. But in our program, we made a little improvement. We use three

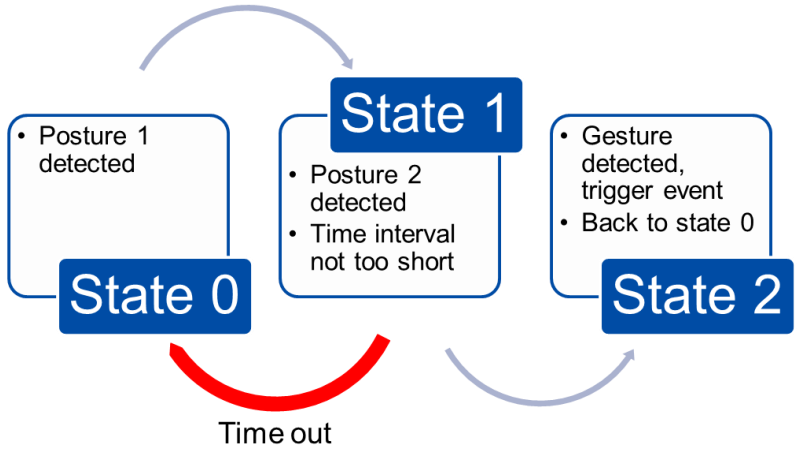
angles to represent the difference of two orientations, which are the angles along the x, y and z axis, respectively. So the threshold is also a Vector3. See `JointMatcher.Match()`.

3. How to detect a posture?

Each `PostureDetector` object contains an array of `JointMatcher` objects. In a simpler manner, when the joints included by the posture are all matched, the posture is considered detected. However, if the thresholds of the joints are too large, the posture will be detected all the time, and false alarms are more likely to occur. It is intuitive to detect the posture only when it is "closest" to the template. So we defined a matching score for each joint and one for each posture. The posture is considered detected only when the matching score of the posture is starting to decrease. See `JointMatcher.GetMatchScore()` and `PostureDetector.Detect()`.

4. How to detect a gesture?

Each `GestureDetector` object contains an array of `PostureDetector` objects. In order to check if the postures included by the gesture are successively detected, we used a simple finite state machine (FSM). The figure below is a simple illustration of a FSM for a gesture with 2 key postures. See `FSM.Update()`.



Two ways of triggering gesture events are defined: 1). only trigger once when the gesture is detected. 2). Continue triggering as long as the user holds the last posture of the gesture. See GestureDetector.`Detect()` and the comments for GestureDetector.m_frameIntv.

5. How to train templates?

After collecting training samples, this is what happens in the automatic training process: for each joint of each posture of a gesture, we have a list of Vector3 objects (see `TemplateTrainer.Train()`). We use the mean of them as the template. The threshold is related to the standard deviation. See `JointMatcher.Train()`. We also need to compute the template of the time intervals between key postures. The method is similar, except for the standard deviation is computed in a somehow robust way. See `TemplateTrainer.TrainTimeIntv()`.

6. Which parts of the algorithm could be further improved?

Well, there are a lot. To name a few:

- The representation of a joint. We used a Vector3 object to represent its orientation. We

also tried to use the “difference of orientation”, see class `JointMatcherR`. However, the attempt is not very successful. We found the latter method not better than the original orientation method. Maybe we could try other representations, such as position, speed, and so on.

- Currently we use a simple template matching method to match joints. This method is simple and fast, but inaccurate. Maybe we could try some classifiers, such as LDA, SVM and so on.

Thanks!