Learning Event Patterns for Gesture Detection

Felix Beier, Nedal Alaqraa, Yuting Lai, Kai-Uwe Sattler
Technische Universität Ilmenau
{first.last}@tu-ilmenau.de

ABSTRACT

Usability often plays a key role when software is brought to market, including clearly structured workflows, the way of presenting information to the user, and, last but not least, how he interacts with the application. In this context, input devices as 3D cameras or (multi-)touch displays became omnipresent in order to define new intuitive ways of user interaction. State-of-the-art systems tightly couple application logic with separate gesture detection components for supported devices. Hard-coded rules or static models obtained by applying machine learning algorithms on many training samples are used in order to robustly detect a pre-defined set of gesture patterns. If possible at all, it becomes difficult to extend these sets with new patterns or to modify existing ones – difficult for both, application developers and end users. Further, adding gesture support for legacy software or for additional devices becomes difficult with this hard-wired approach. In previous research we demonstrated how the database community can contribute to this challenge by leveraging complex event processing on data streams to express gesture patterns. While this declarative approach decouples application logic from gesture detection components, its major drawback was the non-intuitive definition of gesture queries. In this paper, we present an approach that is related to density-based clustering in order to find declarative gesture descriptions using only a few samples. We demonstrate the algorithms on mining definitions for multi-dimensional gestures from the sensor data stream that is delivered by a Microsoft Kinect 3D camera, and provide a way for non-expert users to intuitively customize gesture-controlled user interfaces – even during runtime.

1. INTRODUCTION

Non-traditional input devices gain reasonable attention not only for mobile devices but also for building smart user interfaces to visualize and explore data. Prominent examples are motion sensing devices like Microsoft's Kinect camera or (multi-)touch-screens used in smartphones and tablets.

In our previous works we have demonstrated Kinect-based user interfaces for gesture-controlled interaction with OLAP databases [3] and graph databases [1]. In both prototypes, gestures are identified by detecting complex event patterns, e.g., of the form "if left hand is at position (x, y) and moved within 2 seconds to position $(x + 100, y)$ then gesture = swipe". Such patterns are described as complex event processing (CEP) queries and are executed by a data stream engine on a sensor data stream continuously produced by the Kinect camera. Detected patterns can be easily mapped to application-specific interfaces as navigation operators, e.g., *drill-down* or *pivot* on an OLAP cube, or graph traversal operations in a graph database. Similarly, gestures for touch-based interfaces as described in [4, 5] can be detected.

However, even if most of the gestures used in our demos have shown to be intuitively, defining corresponding CEP queries and their parameters has been an arduous trial-and-error process. Obviously, writing (textual) queries with complex CEP rules is not a user-friendly solution, impeding to fully exploit the sweet spot of declarative gesture definitions: the possibility to quickly define or exchange them without modifying application code. Solving this problem would simplify the work of application interface programmers and, even further, enable end-users to define their own gestures like it is common for keyboard shortcuts to customize controls – increasing the acceptance of such interfaces.

Our contribution is a learning process that requires very few captured gesture samples only in order to derive event patterns which are

- robust enough to detect the intended gesture even if the position or movement of the user differs from the training samples and
- selective enough to distinguish from other patterns.

We present a solution addressing these requirements in the following way:

- We describe a learning approach for deriving event patterns from a sensor data stream and generating CEP-based detection queries automatically.
- We demonstrate how this approach can be used interactively to define new gestures which are detected after very few repetitions by our engine.
- We integrate this approach into a gesture-controlled interface for navigating in a database.

10.5441/002/edbt.2014.70

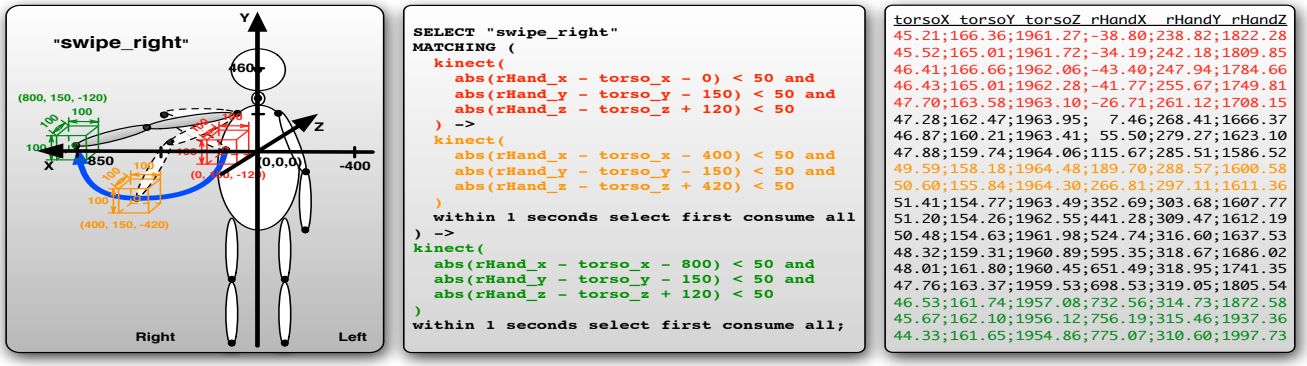

Figure 1: Gesture Detection by Complex Event Processing

2. GESTURE DETECTION BY COMPLEX EVENT PROCESSING

Motion sensing devices like the Kinect camera provide an easy way of detecting body gestures. The development of application interfaces is supported by middleware solutions like OpenNI or the Microsoft Kinect SDK which implement already most of the tasks of detecting and tracking body skeletons. Based on this, an application can receive a continuous stream of sensor readings describing the current position of certain skeleton joints in a 3D coordinate space, e.g., as shown on the right in Fig. 1.

However, implementing gestures beyond predefined standard patterns is still a tedious task. In [3] we have already demonstrated how to simplify this by using our data stream management system AnduIN with CEP functionalities.

For the purpose of gesture detection from a sequence of sensor readings we apply the `match` operator of our AnduIN system which implements pattern matching using an NFA. An example of such a pattern detection query is given in the center of Fig. 1. Events are certain positions of a body part (in this example the right hand relative to the position of the torso). These poses are described by spatial regions around the user's body (Fig. 1 left). Based on this, gestures are defined by a sequence of events denoted by the `->` operator together with optional time constraints (in this case within 1 second). As soon as the current input data matches this pattern, a result tuple is produced (comprising the string value "`swipe_right`" in our example) which can be used to trigger arbitrary actions in any listening application.

Thus, the main goal of the work described here, is to derive such queries from example data.

3. LEARNING EVENT PATTERNS

An overview of the whole learning workflow is illustrated in Fig. 2. The user interacts with the tool via a graphical user interface that guides him through the learning process, providing visual feedback during the steps. Kinect measures are streamed through our AnduIN CEP engine for (i) a touchless GUI control and (ii) applying necessary data transformations on-the-fly when new training samples are recorded. After transforming the sensor data in a user-independent format, the sample data is stored in a database for further processing and manual debugging. A density-based data mining algorithm is applied on each gesture sample to extract characteristic points describing the gesture path. Further samples can be added to incrementally improve the results until the user is satisfied with the mined CEP patterns. Therefore, the mining algorithm is applied to each sample separately and partial results are merged to the final gesture description. Usually, 3-5 samples are sufficient to achieve acceptable results. All gesture patterns are stored in a database for an optional post-processing step where patterns can be (i) simplified to improve detection times and (ii) cross-checked to avoid "overlaps". To deploy gestures, CEP queries are generated and used in AnduIN to let the user test and verify detection accuracy. During this process he is guided with visualizations which support identifying reasons in case of detection problems. Since gestures are described through parameters directly in a user-oriented coordinate space, manual fine tuning is easily possible without a separate re-learning by simply adapting the queries.

3.1 User Interaction

To ease the definition of new gestures, we carefully designed the way users provides input to the program and how generated information is visualized. While it is acceptable to start the learning process through console or GUI in order to provide necessary parameter settings, this approach is impractical when gesture samples shall be recorded. Therefore, we make use of pre-defined, but configurable gestures to control the learning tool itself. When the user wants to record a new sample for a gesture, he triggers the process with a *wave* gesture. In order to avoid false measurements right after the control gesture has been detected, the user has to move to the starting pose of the gesture he would like to perform. The actual recording is triggered after the user did not move for some time and lasts until the user stops at the end pose. Everything in between is regarded as part of the gesture and forwarded to the learning component. This way, the user can record multiple samples consecutively without having to leave his position in front of the camera.

A *swipe* gesture with both hands is used to finalize the learning process and start the testing phase. Therefore, the gesture CEP queries are generated, deployed in AnduIN, and the user can verify if his movements are correctly detected. To visually guide him in this process and support him to identify possible reasons why a movement was not detected, the Kinect video stream is displayed, a 3D model is animated performing the movement, and overlay gesture information is visualized, e.g., the 3D windows generated during the learning phase and paths of tracked skeleton joints.

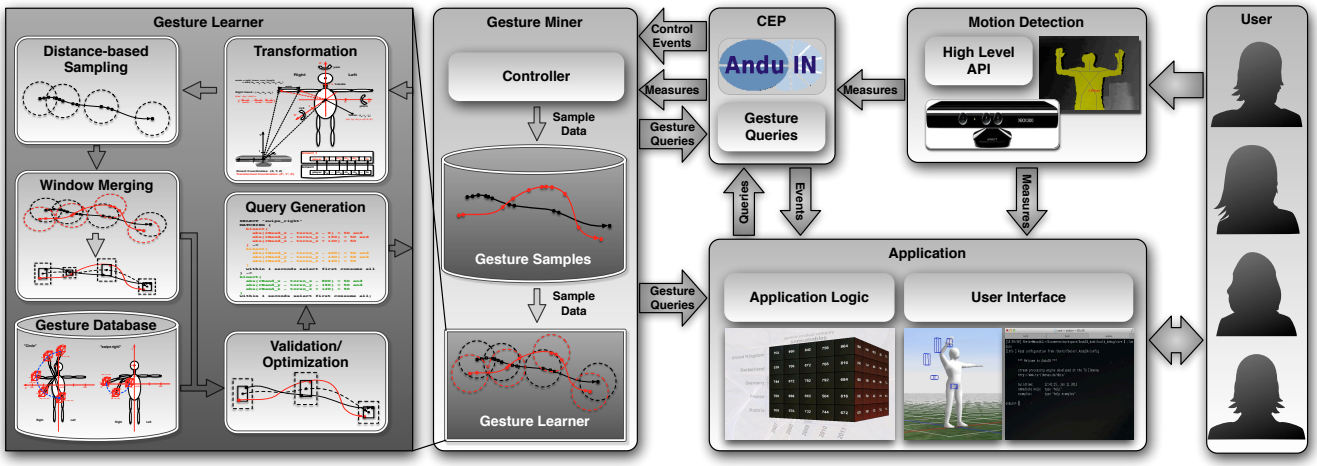

Figure 2: Gesture Learning Process Overview

3.2 Data Transformation

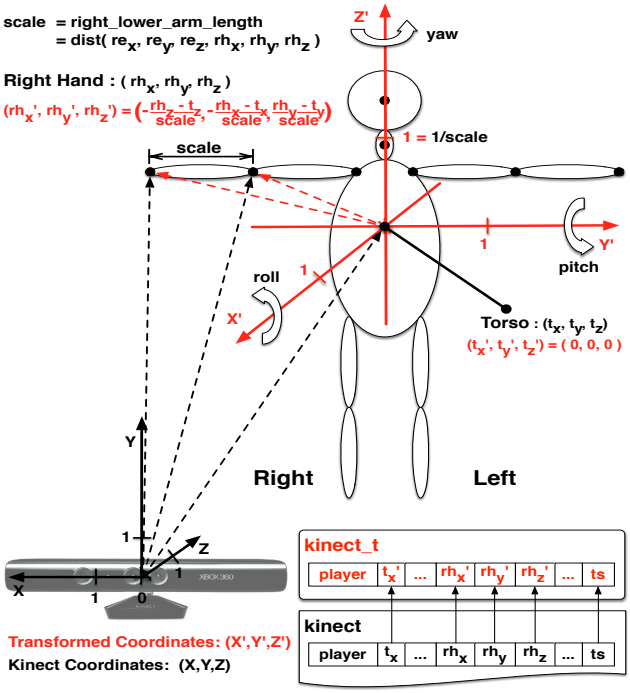

Figure 3: Data Transformation

A challenge for expressing gestures robustly is finding a suitable coordinate system to define CEP patterns that would detect the same movements even if different people perform them. Skeleton joint coordinates that are extracted from the camera video stream are in a camera-related coordinate system illustrated in black in Fig. 3.

To get **position-invariance**, all joints coordinates are shifted by the torso with subtracting its current (t_x, t_y, t_z)-position from each joint, turning the torso into the new origin. This enables gesture detection even if the user moves relatively to the camera. Further, the coordinate axis are rotated as illustrated in red in Fig. 3. The user's viewing direction becomes the new X-axis, leading to a *East-North-Up* (ENU) ground reference frame as it is used for land vehicles.

The calculation of Roll-Pitch-Yaw (RPY) angles defined in this system were implemented as user defined operators in AnduIN. They can be used to easily express movements using any kind of rotations, e.g., a *wave* gesture.

scale-invariance is required to detect the same movement performed by users of different height. Therefore, all coordinates are scaled by the right forearm length, assuming that tall people have longer arms than smaller people, which turned out to be a good approach in our experiments when testing the same gestures with children and adults. The scale factor is calculated as the Euclidean distance between the right hand and right elbow and remains constant no matter how the user is oriented in front of the camera.

Note that for applying all transformations, only a single step needs to be performed on the incoming data stream. Hence, we defined a `kinect_t` view letting AnduIN calculate all coordinates on-the-fly. Other transformations are possible with this declarative approach, e.g., expressing joints with Euler angles, but are out of the scope of this paper.

3.3 Gesture Learning

The key idea behind our approach for learning gestures is the interpretation of a gesture as a sequence of poses. As discussed in Sec. 2, each pose can be described by spatial regions where involved skeleton joints are located. Currently, we express these regions as multi-dimensional rectangles ("windows"), having a center point determined by all (x, y, z) joint coordinates and a width in each dimension representing possible deviations. Other representations are possible but we've chosen these fixed rectangular boundaries since they can be easily expressed by range predicates. This facilitates query generation and the visualization of these "detection conditions" to allow simple debugging as well as manual parameter adaptions later on. Poses are combined by `sequence` CEP operators to express the gesture path.

3.3.1 *Distance-based Sampling*

Since the Kinect sensor stream delivers tuples at a rate of ≈30 Hz, taking each measure as separate pose is impractical for two reasons: First, using many CEP patterns for describing one gesture increases detection complexity. Second, gesture samples are overfitted, leading to low detection rates for slightly different movements expressing the same gesture.

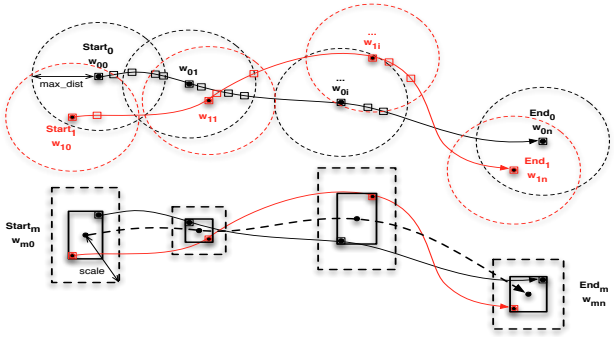

Figure 4: Gesture Learning

Therefore, we implemented a distance-based sampling technique that is comparable to density-based clustering [2]. At the top of Fig. 4, two gesture paths are illustrated. Each tuple is visualized as small rectangle. The goal is to obtain window definitions as shown at the bottom of Fig. 4.

Similar points are clustered together in one window based on a distance metric, i.e., a function that is calculated between two points. The first tuple is used as initial cluster centroid and as reference for calculating future distances. A new window is added and used as next reference as soon as a new measure point differs significantly from the last reference cluster, i.e., its distance exceeds a `max_dist` threshold.

The distance function is configurable to express several gesture semantics, e.g., the Euclidean distance can be used to express spatial differences between successive poses, or metrics like "*every x tuples*" can be used for time-based constraints. Distance thresholds are automatically calculated relative to the whole gesture path, e.g., "*at least x% of the total deviation observed*".

3.3.2 Window Merging

When the same gesture is repeated multiple times, recorded samples usually differ slightly. Therefore, relevant clusters that have been separately extracted for all paths have to be merged in order to obtain a final gesture description which is general enough to detect all of them. To extract rectangular spatial regions, we calculate minimal bounding rectangles (MBR)s around all cluster centroids with the same sequence number (bottom of Fig. 4). This step can be executed incrementally and is useful for detecting situations where a new sample differs too much from previously recorded ones, allowing us to issue a warning in this situation. To generalize gesture patterns further, another scaling step can be performed by increasing the rectangles' width in each dimension. But scaling them too much introduces the *overlapping problem*, i.e., patterns of different gestures detect the same movement. This problem usually reveals fast when mined CEP patterns are visualized during the testing phase. It can be easily solved by manually adding additional constraints to generated queries that separate conflicting gestures.

3.3.3 Outlook: Validation & Optimization

An optional post processing step can be applied to check final gesture patterns. Intersection tests can be performed on windows to determine if the overlap problem occurs. Further, patterns can be optimized, e.g., by merging windows to decrease the detection effort or by eliminating certain coordinates that are not relevant for the recorded gesture.

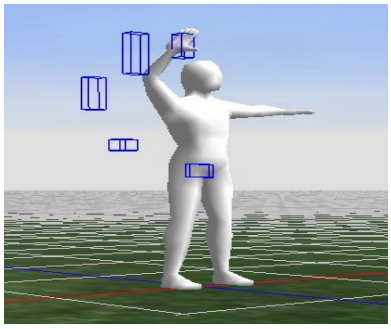

Figure 5: 3D Model

3.3.4 Query Generation

Finally, a query has to be generated for the CEP engine. Therefore, the following range predicates are generated for each pose's MBR: j denotes the joint involved in the gesture, i its possible coordinates, *center* the MBR's center point, and *width* the MBR's width in that coordinate direction:

$$\bigwedge_{\substack{j \in joints, \\ i \in \{x,y,z\}}} abs(center_{j,i} - coord_{j,i}) < width_{j,i}$$

Predicates of different poses are joined with nested `sequence (->)` operators where the first one is directly applied on the transformed stream `kinect_t`. The output tuple sent to the application on gesture detection is configurable. Usually it contains the gesture name and – if required – some measures that are calculated directly on the stream during the detection process, e.g., joint positions or movement speeds.

4. DEMONSTRATION

During the demo session we will bring a computer equipped with a Kinect camera and running graphical applications that can be controlled by users via gesture commands for navigating through a graph and an OLAP database [1, 3]. Demo visitors will get the opportunity to define new gestures using a tool that implements our presented approach for learning gesture definitions. The tool provides insight in each processing step and uses an animated 3D human body model (Fig. 5) for visualizing mined gesture patterns. Gesture parameters can be manually adjusted after the learning process which is reflected in the model to render the effects graspable. Newly defined gestures can be instantly tested by each visitor by exchanging the applications' pre-defined navigation operations during runtime, demonstrating the full flexibility of the declarative gesture detection approach.

5. REFERENCES

[1] F. Beier, S. Baumann, S. Hagedorn, H. Betz, and T. Wagner. Gesture-Based Navigation in Graph Databases – The Kevin Bacon Game –. *BTW '13*, 2013.

[2] M. Ester, H. peter Kriegel, J. S, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, 1996.

[3] S. Hirte, E. Schubert, A. Seifert, S. Baumann, D. Klan, and K.-U. Sattler. Data3 - A Kinect Interface for OLAP using Complex Event Processing. In *ICDE*, April 2012. Video: http://youtu.be/DROXI0_wDRM.

[4] S. Idreos and E. Liarou. dbTouch: Analytics at your Fingertips. In *CIDR*, 2013.

[5] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *Proc.of the VLDB Endowment*, 7(4), 2013.