

## Research Article

# Modular Middleware for Gestural Data and Devices Management

**Fabrizio Milazzo,<sup>1</sup> Vito Gentile,<sup>1</sup> Giuseppe Vitello,<sup>1</sup>  
Antonio Gentile,<sup>1,2</sup> and Salvatore Sorce<sup>1,2</sup>**

<sup>1</sup>*Ubiquitous Systems and Interfaces Group, Dipartimento dell'Innovazione Industriale e Digitale (DIID),  
Università degli Studi di Palermo, Viale Delle Scienze, Edificio 6, 90128 Palermo, Italy*

<sup>2</sup>*InformAmuse Srl, Academic Spin-Off, Via Nunzio Morello 20, 90144 Palermo, Italy*

Correspondence should be addressed to Fabrizio Milazzo; [fabrizio.milazzo@unipa.it](mailto:fabrizio.milazzo@unipa.it)

Received 4 November 2016; Revised 27 February 2017; Accepted 6 March 2017; Published 18 May 2017

Academic Editor: Jose R. Martinez-De-Dios

Copyright © 2017 Fabrizio Milazzo et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In the last few years, the use of gestural data has become a key enabler for human-computer interaction (HCI) applications. The growing diffusion of low-cost acquisition devices has thus led to the development of a class of middleware aimed at ensuring a fast and easy integration of such devices within the actual HCI applications. The purpose of this paper is to present a modular middleware for gestural data and devices management. First, we describe a brief review of the state of the art of similar middleware. Then, we discuss the proposed architecture and the motivation behind its design choices. Finally, we present a use case aimed at demonstrating the potential uses as well as the limitations of our middleware.

## 1. Introduction

Many works in psychology and social science have shown the relevance of gestures as a mean for communications among human beings [1]. This is probably one of the main reasons for the recent investigations on the use of midair (or touchless) gestures as a mean for easing human-computer interactions [2, 3].

The touchless nature of gestures makes it possible to think about new modalities for accessing information and controlling the surrounding environment. For instance, ambient intelligence and smart-home systems can use gesture recognition to control lighting, temperature, cooling fans, and so on, by simply recognizing hands and/or body movements [4–7]. Gesture-based interaction may also come in help to physically impaired people who may have a chance to interact with the surrounding environment or with their own aids, such as a wheelchair [8]. Gestural interfaces can also be useful for enhancing the protection of interaction devices against vandalism [9].

In our opinion, the today's gesture interaction can be considered a research branch of the broad Ubiquitous

Computing (UC) and Internet of Things (IoT) paradigms, which obviously includes the modalities with which the users interact with their surrounding environment and in particular the ways they access sensors and visualization devices [10, 11]. In this sense, recent works in HCI have focused their attention on the development of a class of middleware specifically aimed at the management of gestural data and the communication with the related input devices [12, 13].

In the ideal case, gesture management middleware should offer a fast solution to the problem of connecting the input devices to the end-user gesture-based applications. In particular, it should act as a transparent layer able to manage communication towards the devices, to unify heterogeneous gestural data, and to provide easy access to the same data through software services.

To the best of our knowledge, very few gesture management middleware solutions are able to support the above desiderata. The goal of this paper is to propose novel modular middleware aimed at the management of gestural data. It is able to manage different acquisition devices and to provide a unified model for the heterogeneous input data. Moreover,

it allows for a transparent communication between the end-user applications and multiple input devices and offers a web services based access to the gathered data and to the gesture management services.

The rest of the paper is arranged as follows: Section 2 presents a brief review of the state of the art of the gesture management middleware; Section 3 presents the architecture of the proposed middleware as well as the provided features; Section 4 highlights some technical limitations in the use of the middleware and evaluates its performance in a real-world scenario. Finally, Section 5 discusses some possible improvements of our proposal, along with the future works.

## 2. Related Works

The purpose of this section is to review a list of selected relevant solutions for gestural data and devices management. We overview their history and the technical improvements over the time; we also arrange a brief comparison of the main features of the selected reviewed solutions and our proposed one.

According to our analysis of the relevant literature, we have recognized some peculiar features for classifying gesture management middleware:

- (i) *Managed devices*: this feature indicates which devices the middleware is able to manage (2D devices such as touchscreen and mouse or 3D devices such as RGB and depth cameras).
- (ii) *Abstraction*: this feature indicates whether the middleware provides a unified access to all the managed devices.
- (iii) *Gestural data processing*: this feature indicates which operations can be carried out with input gestural data, for example, gestures management (recording, retrieving) and gestures recognition.
- (iv) *N-by-M communication*: this feature indicates whether the middleware is able to manage multiple connected devices and multiple end-user gestural applications.

The first notable gesture management middleware is *iGesture* [14] (2007). It is written in the Java programming language and is able to manage only 2D devices such as digital pens and mice. Devices communicate within the middleware by means of a device abstraction layer. The recognition capabilities are limited to few hard-coded gestures. Moreover, the middleware can manage only one device at a time.

The work proposed in [15] consists in an improvement of the *iGesture* middleware and implements support for 3D devices. As the authors claim, it is possible to connect Wii Motion and Wii Remote simultaneously for gesture acquisition. The gesture recognition task is carried out by means of Dynamic Time Warping [16].

*AQUA-G* [17] is another gesture management middleware which offers support for 2D/3D acquisition devices. It has introduced the concept of three-layered architecture, which can be seen as a reference for the gesture management middleware. The architecture includes the *input* layer, which

allows connecting different input devices and accesses their input streams; the *gesture processing* layer, which implements recognition and learning services; the *application* layer, which provides some ready-to-use services for the end-user applications (e.g., stream access, gesture recording, and playing back).

The *ARAMIS* [18] framework (2011) was written with the purpose of providing gesture management software for the development of pervasive computing applications. It differs from the previous works as it introduced *data-synchronization* and *data-fusion*: this ensures that data coming from different input device classes (e.g., environmental and wearable ones) follow the correct time order. A further extension of *ARAMIS*, named *FEOGARM*, has been described in [19]. It is a modular framework for multimodal and multisensor applications, developed to evaluate the performance of different gesture recognition algorithms. This work proposes a novel gesture segmentation method, which operates on the raw data and, as a result, speeds up the process of dataset collection and labeling. Anyway, it appears to be a closed system and it is not clear if and how it is possible to add new classes of input devices.

One solution for pervasive computing applications is proposed in the work by Chaczko et al. [20] (2015). It was developed with the aim of supporting *Kinect-like devices*, that is, devices able to provide RGB, depth, and skeletal information [21]. Data can be managed via a web-service based access [22, 23]. The middleware has been tested in an environment which included a Microsoft Kinect, a Leap Motion, and a Thalmic Labs Myo. One key ability is to recognize and select the user's nearest device while keeping the farther ones in an idle state; this allows lowering power consumption and network bandwidth.

A more recent work, proposed by Moreno et al. [24], has demonstrated real time capabilities for the management of the RGB, depth, and skeleton channels coming from a Microsoft Kinect. The channels can be accessed by means of web sockets (which allow many client applications to simultaneously read the data). The main flaw of this work is its inability to work with other gestural input devices and the impossibility of recording and playing back the input data.

In Table 1 we have summarized the main features offered by both the revised middleware and our proposal. Our middleware offers an all-in-one solution for easing the development of gesture interaction applications. It provides a unified model for all the 2D/3D supported gestural inputs. Moreover, our framework can be easily expanded by adding new classes of input devices. It is designed to support *N-by-M* communication, that is, managing *N* different connected devices and sending their data towards *M* end-user applications, with some limitations (see Section 4 for more details). It offers the possibility of acquiring, reading, and editing new gestural datasets, by means of a networked database (no matter if it is local or remote). We also provide the possibility of using different recognition algorithms for the gestures, in order to let users test the recognition performance for the acquired datasets. All the middleware features are accessible by dedicated web services, which implement them as REST APIs.

TABLE 1: Main features of the middleware for gesture interaction.

Approach	2D/3D devices	Abstraction layer	Gestures management	Gestures recognition	N-by-M communication
<i>iGesture</i> [14]	✓X	X	X	✓	X
<i>AQUA-G</i> [17]	✓✓	✓	X	✓	✓
Bas [15]	✓✓	X	X	✓	X
<i>ARAMIS</i> [18]	✓✓	✓	X	✓	✓
<i>FEOGARM</i> [19]	✓✓	✓	X	✓	X
Chaczko et al. [20]	X✓	✓	✓	X	X
Moreno et al. [24]	X✓	X	X	X	✓
<i>Our proposal</i>	✓✓	✓	✓	✓	✓

It is worth noting that many general-purpose solutions have been described in prior works (especially in the area of robotics), which in principle may implement most of the features proposed by our middleware. Probably, the most known one among these frameworks is the Robot Operating System (ROS) [25]. It is a modular solution which is thought to make the development of robotic applications easy. It provides support for several input sensors (including the classical gestural input ones such as Kinect and Leap Motion Controller) and some features for recording and playing back data. However, since ROS was not developed to target gesture interaction applications, its use in this field is quite limited by some issues. First, it lacks a unified representation model for gestures; also, the data storage system is based on the so-called “bag-files,” which necessarily introduces unwanted delays in the record/retrieval process; finally, as a design choice, our middleware accesses data streams natively, by using only the official device drivers, whereas in ROS this kind of support is only partial and not at the native level, as stated in the official ROS website [26].

### 3. Middleware Overview

The proposed middleware is aimed at supplying end-user gesture interaction applications with a set of facilities, ranging from transparent communication towards gestural input devices, to data acquisition, retrieval, and recognition. Its architecture expands upon three layers, as shown in Figure 1.

The *Physical* layer communicates with the input devices and is mainly responsible for providing access to their data streams. The *Processing* layer implements all the gesture data management features, including acquisition, retrieval, and recognition of each stream coming from the Physical layer. Finally, the *Service* layer is responsible for the provision of services to be used by the end-user gesture interaction application. At this purpose, the services will be provided as a set of RESTful APIs allowing a remote, easy, and protected access to the system.

The three layers are implemented by six main components, which serve the following tasks:

- (i) *Device*: the software representation of an input device
- (ii) *Communication*: implementing services to exchange data among layers
- (iii) *Gesture*: implementing gesture recognition features

(iv) *Database*: representing the software interface towards the actual storage

(v) *Controller*: exposing facilities for end-users gestural interaction applications

(vi) *Model*: containing the description of the data flowing in/out the middleware.

The configuration of the above components is written in a specific file described in Section 3.7.

The following sections put the emphasis on some prominent implementation details and explain how the features provided by our middleware and claimed in Table 1 are achieved. The architecture is conceived for Object Oriented Programming (OOP). For this reason, we will make use of UML diagrams for describing its components implementation.

**3.1. Device.** The Device component actually implements the Physical layer. It is in charge of managing each of the connected devices and of relaying their data to the upper (Processing) layer. Figure 2 depicts its composing classes, namely, the *DevicesManager* and *DeviceDriver*.

The *DevicesManager* class is a sort of devices container which is able to start/stop them as well as enable/disable their data streams. The *DeviceDriver* class, in turn, is a generic representation of what a gesture device should be capable of and is in charge of communicating with the native drivers of the related input device. The *dev\_state* variable represents its current state which can be one among the following: (i) ready, meaning the device is attached and ready to be started; (ii) shared, meaning the access to the device streams will be allowed for more than one end-user application; (iii) exclusive, meaning the access to device streams is restricted to just one application and rejects any new request for accessing its data streams.

Since each device has a variable number of manageable streams, then each driver contains a variable named *buses* which associates a stream name (i.e., a string) to a *Bus* class instance. As an example, Kinect can have three manageable data streams (one for RGB, another for the depth map, and the last for skeletal data), whereas a Leap Motion Controller is able to produce only depth and skeletal data.

New devices can be easily integrated within the middleware. In order to do so, it will be sufficient to implement an

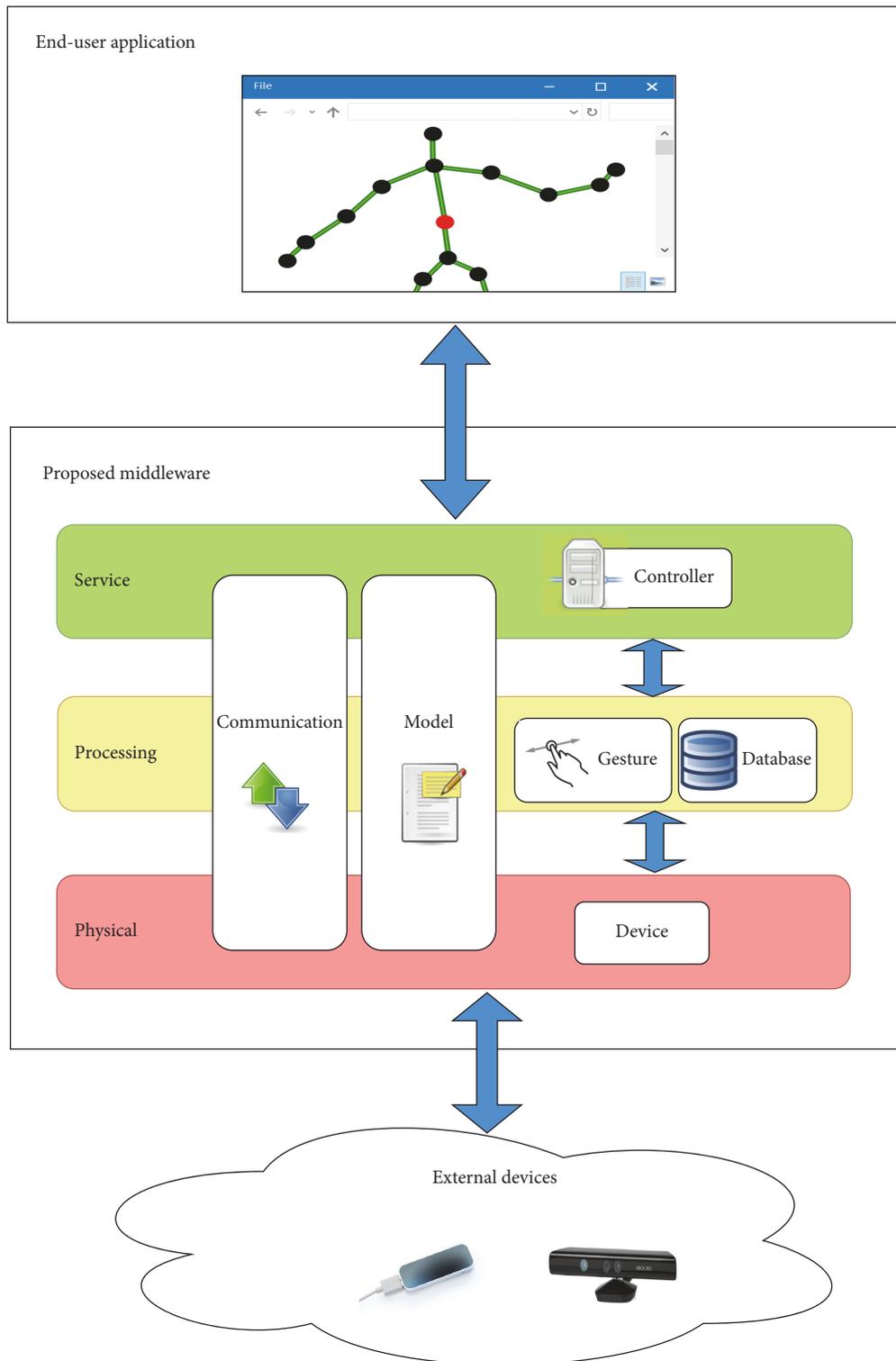


FIGURE 1: The layered structure of the proposed middleware.

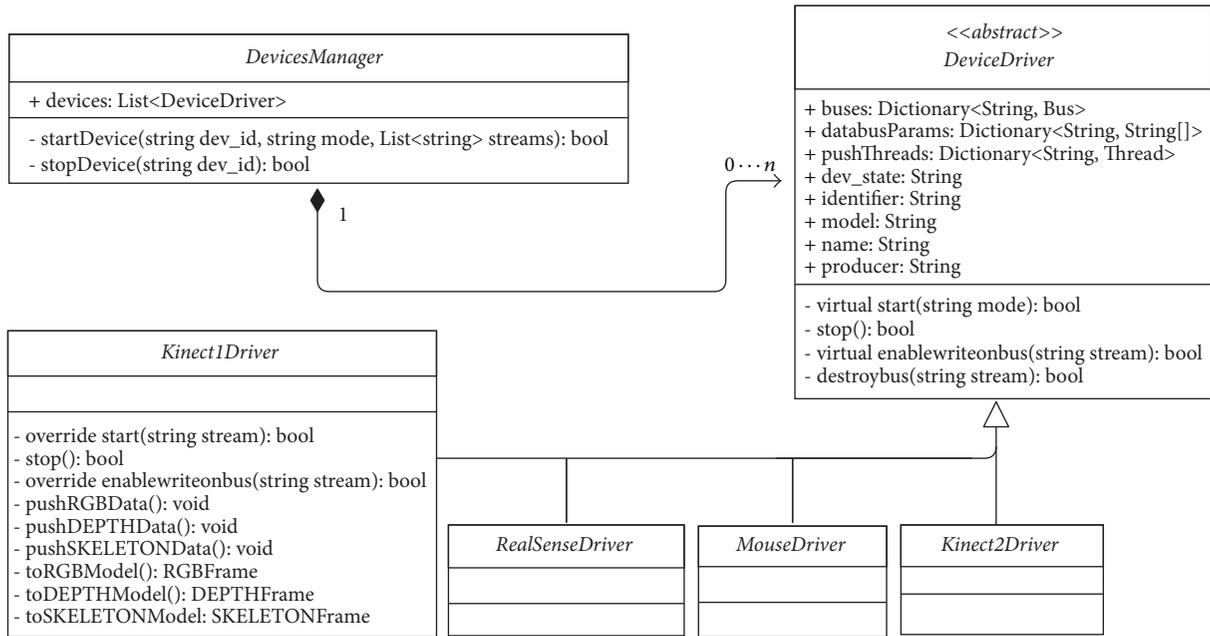


FIGURE 2: UML diagram of the device component.

actual driver which inherits from the (abstract) *DeviceDriver* class.

The following methods must be implemented:

- (i) *start(start\_mode)*: it tries to connect to the device, checks whether the device can be started in the selected model, and returns a Boolean as a result.
- (ii) *enablewriteonbus(stream)*: it checks whether the requested stream is managed and, if yes, starts a thread running the *pushStreamData* method.
- (iii) *pushStreamData*: it is the method which actually sends the stream data to the output bus. Clearly, the driver must implement one method for each stream it is intended to manage.
- (iv) *toStreamModel*: this method converts the data coming from a given device stream (necessarily device dependent) into the standard and unified representation adopted in the middleware (see Section 3.6 for details).

Currently, our middleware implements the drivers for computer mice (2D device) and some common 3D gestural devices such as Kinect v1, Kinect v2, and the Intel RealSense Camera (model F200).

**3.2. Communication.** This component implements the channels devised for hosting data gathered from input devices. The *Bus* class (abstract) is very similar to an actual “bus” and is characterized by an address, that is, its URI, and the *ReceiveAndOutput* method, which is in charge of writing data in the channel. Its use is straightforward: whenever data is read from the input device, then the *pushStreamData* method of the *DeviceDriver* class selects an appropriate bus and calls

*ReceiveAndOutput* in order to make the data available to the end-user applications.

This component is very important as it makes the way data are gathered from the input device and relayed to the outside transparent. The current implementation of the middleware offers *WebSocket* and *File* buses. As an example, it will be possible to configure a Kinect to send its data to a file and a RealSense sending data to a web socket connection.

**3.3. Gesture.** The Gesture component implements gesture recognition features. It is able to train different gesture recognizers based upon the gestural data stored in the database. As shown in Figure 3, the component is composed of four classes: (i) *RecognitionManager*, (ii) *Recognizer*, (iii) *ClassifierStrategy*, and (iv) *HMMStrategy*.

The *RecognitionManager* is the container for all the recognizers instances. Its *init* method accepts as input the list of the data streams to be recognized as well as a confidence threshold to be used for recognition. The *init* method fills a list of the *Recognizers* available for the requested data streams. We want to point out that there is a one-to-one mapping between recognizers and device streams. For instance, there will be one recognizer for the pair (Kinect1, SKELETON) and another one for (Kinect2, SKELETON).

The *start/stopRecognizeDevice* method accepts as input parameters the *dev\_id*, that is, the device identifier and the name of the *stream* to be recognized; the *saveRecognizer* method allows saving a *Recognizer* instance into a binary format. The *trainModel* method allows training a classifier based on the data recorded in the database. In particular, the method accepts as input the strategy class name (i.e., which classifier should be used for recognition), its confidence threshold, the device identifier, and the stream name.

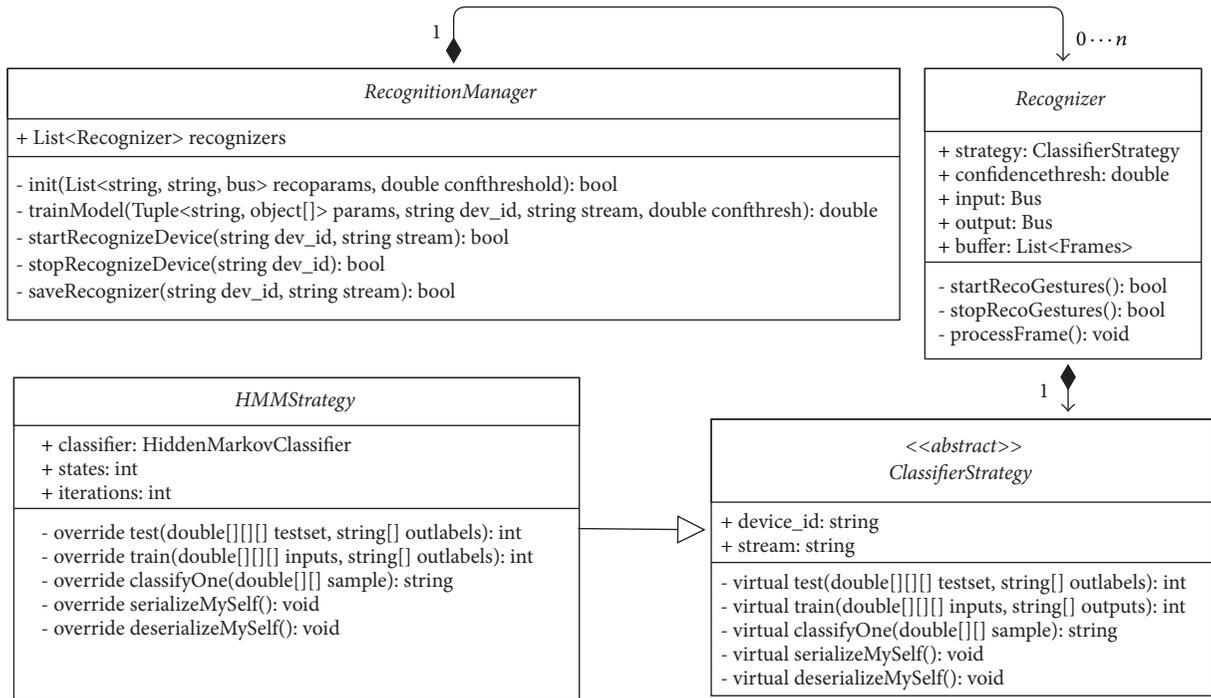


FIGURE 3: UML diagram of the gesture component.

By following the principles of the *Strategy* design pattern [27], each *Recognizer* owns an instance of the *ClassifierStrategy* class, which actually implements the classifier. The motivation behind such choice is to allow end-users to train and test different recognition algorithms over the same gestural data.

Currently, the middleware implements only a Hidden Markov Model (HMM) classifier for training/recognizing gestures; anyway, it is very easy to extend the available recognition algorithms by inheriting from the *ClassifierStrategy* class. In particular, each actual *Strategy* must implement the *train*, *test*, and *classify* methods. Since the classifiers must be stored within the physical database, they must implement the *serializeMySelf* and *deserializeMySelf* methods, which encode/decode the recognizer into/from a convenient binary format.

**3.4. Database.** The Database component is an abstract representation of the physical storage. It is in charge of providing recording and retrieval features for the gestural data. The current implementation relies on a physical MongoDB database [28], a NoSQL solution which uses a JSON-like language for representing query results, stored as collections of documents.

The database manages the following four collections:

- (i) *Device*: it stores the `identifier`, `name`, `model`, and `producer` of the known devices.
- (ii) *Streams*: it contains `identifier` and `name` of all the known streams.
- (iii) *Gestures*: each record represents a sequence of gestural data and is characterized by a `tag` (the gesture

name), `device_id` (the input device), `stream` (the input stream), and the array `frames` (the gestural data sequence).

- (iv) *Strategies*: it contains the binary representation of the classifiers trained by the *RecognitionManager*.

**3.5. Controller.** The role of the *Controller* component is to work as a wrapper for the entire middleware. Any application built on top of our middleware must use its services exposed as REST APIs. The available web services are highlighted in Box 1.

The *List Device* service returns the list of the working devices, together with their accessible streams. The *Start Device* service accepts as input the device identifier, a list of streams to be read, and the working mode of the device; the answer specifies whether the device is available in the current chosen mode and the list of the URI to access the requested streams.

The *Start Recognition* service is characterized by a device identifier and the streams onto recognition must be activated. The answer indicates (i) whether the recognition service can be activated for the device (it must be working in shared mode); (ii) whether there is a *Recognizer* available for each of the requested streams and, if the case, (iii) the URI to access the tags of the recognized gestures. A negative answer means that recognizers have not been previously trained for the requested device-stream pairs.

The *Show Gesture* service provides retrieval from database. The resulting gestures can be filtered by device, stream type, gesture tag, and position in the output collection. The *Delete Gesture* service needs only the gesture identifier parameter.

```

List Device:
http://{server_url}/devices
Start Device:
http://{server_url}/devices/start/?id={dev_id}&streamsName={stream_1,...}
&mode={exclusive|shared}
Stop Device:
http://{server_url}/devices/stop/?id={dev_id}
Start Recognition:
http://{server_url}/devices/startRecognition/?id={dev_id}
&stream={stream}&mode=shared
Stop Recognition:
http://{server_url}/devices/stopRecognition/?id={dev_id}&stream={stream}
Show Gesture:
http://{server_url}/gestures/show/?id={dev_id}&stream={stream}
&tag={gesture_tag}&pos={cursor_pos}
Delete Gesture:
http://{server_url}/gestures/delete/?id={gesture_id}
Save Gesture (HTTP POST):
[type: post, url: http://{server_url}/gestures/save,
contentType: application/json, data: gesture]
Update Gesture (HTTP POST):
[type: post, url: http://{server_url}/gestures/update,
contentType: application/json, data: {gesture_id, gesture}]

```

Box 1: The service strings exposed by the REST API Controller.

```

{
  _ID: string
  NAME: string
  MODEL: string
  PRODUCER: string
}

```

Box 2: Device class.

```

{
  _ID: string
  NAME: string
}

```

Box 3: Stream class.

While the aforementioned services were implemented as simple HTTP GET, the *Save/Update Gesture* services require the use of the HTTP POST method. This is because, in such particular case, the invoker must send the gestural data frames to be written in the database. The format of the gestures accepted by these services is specified in Section 3.6.

**3.6. Model.** The purpose of this section is to describe the format of the data managed within the middleware. The *Model* component is implemented as a set of classes containing the metadata useful for managing (i) devices, (ii) streams, (iii) strategies, and (iv) gesture data. The aim of such classes is to provide a link between the unstructured data, physically stored in the database, and the middleware objects. The *Device* class stores the four fields (see Box 2).

The *Stream* class contains only two values (see Box 3).

The *Strategy* class contains all the metadata needed for instantiating a strategy previously trained over gestural data. It holds a type (currently it can be only the HMMStrategy), the device identifier, the stream name, and its serialized

```

{
  _ID: string
  TYPE: string
  DEV_ID: string
  STREAM: string
  SERIALIZED_REPRESENTATION: byte[]
}

```

Box 4: Strategy class.

representation which will be converted (at run-time) into a real classifier object (see Box 4).

The *Gesture* class contains metadata devised for the description of a physical gesture, in particular an identifier, a gesture tag (i.e., a label indicating the gesture name), the source device, and stream. Furthermore, a gesture clearly holds a sequence of frames; the latter are composed of a timestamp and an array of joints (i.e., the connection points between two bones). Each joint is identified by a name, its position in the world ( $x, y, z$ ), the position in the depth

```

GESTURE:
{
  _ID: string
  TAG: string
  DEV_ID: string
  STREAM: string
  FRAMES: frame_type[]
}
FRAME:
{
  JOINTS: joint_type[]
  TIMESTAMP: long
}
JOINT:
{
  NAME: string
  WORLD_POSITION: float[] // x, y, z coordinates
  DEPTH_POSITION: float[] // x, y
  ORIENTATION: float[] // a, b, c, w angles
  VELOCITY: float[] // vx, vy, vz
  CONFIDENCE: float
  STATE: string
}

```

Box 5: Gesture class.

map  $(x, y)$ , the local orientation (i.e., a quaternion representing rotational angles), the space velocity, a confidence value (between 0 and 1), and its state (i.e., “tracked” or “not\_tracked”) (see Box 5).

**3.7. Configuration File.** The configuration file is written in JSON [29] and holds the information needed to start up middleware. The content of such file is loaded just before its execution; then, the components are configured accordingly. A typical scheme for the configuration file is shown in Box 6. First of all, the information about the location of the physical database is provided. Connection is implemented via a network protocol, so it will be sufficient to provide the database name, the IP address of the hosting machine, and the process port.

Furthermore, the information about the devices to be controlled at run-time is written in the *Devices* section. In particular, each device is characterized by an identifier, the *Driver* class which should be used for connecting to the device, and the managed streams; each stream, in its turn, will be characterized by its name and the bus to be used for writing output data, and the IP address/port over the output data will be made available. The *Controller* section contains the IP address/port to access the REST APIs. The *RecognitionManager* section provides all the information needed for loading *Recognizers*. We want to point out that each *Recognizer* is associated with a bus and that such bus can be accessed in order to read which gestures have been recognized. At this purpose, the section contains a confidence threshold (currently the same confidence threshold will be shared among all the recognizers) and an array containing for each element a device identifier, the stream name, the bus class, and the IP address/port address over which the recognized gesture labels will be available.

## 4. Middleware Evaluation

The purpose of this section is to evaluate the computational requirements of the proposed middleware. First, we will describe some technical hardware limitations which users must be aware of before running the middleware. Then we will report the performance of the middleware obtained in a real deployment.

**4.1. Hardware Limitations.** The proposed middleware can be replicated on different machines, so, in principle, there are no particular limitations on the number of supported devices (say  $N$ ) or requesting end-user applications (say  $M$ ). Anyway, some important considerations should be done in the very basic case where a middleware instance is running on a single machine.

The hardware limitations are mainly due to the machine bandwidth and the CPU resources. Reading from a specific device stream and writing to a specific output stream are the most bandwidth demanding operations; on the contrary, executing gesture recognition algorithms is the heaviest CPU demanding task. For clarity’s sake, Table 2 summarizes the bandwidth requirements for each of the supported device streams, as well as the number of instructions required by the recognition algorithm (currently, the framework supports only the Hidden Markov Model recognizer).

The columns report, respectively, the name of the device, the stream, the available resolution at 30 fps, the read/write required bandwidth (Mbps), the number of millions of required instructions per second (MIPS) by one Hidden Markov Model, and some particular hardware notes. The data reported in the table was taken by the official documentation of the Kinect v1 [30], Kinect v2 [31], and RealSense F200 [32].

```

{
  "DATABASE":{
    "dbname":"database name",
    "ip":"ip address",
    "port":"port number"
  },
  "CONTROLLER":{
    "ip":"ip address",
    "port":"port number"
  },
  "DEVICES":[
    {
      "id":"id of the device",
      "driverclass":"name of the controlling driver",
      "streams":[
        {
          "name":"stream name",
          "busclass":"the bus for writing output",
          "ip":"ip address of the output bus",
          "port":"port number"
        }
      ]
    }
  ],
  "RECOGNITIONMANAGER":{
    "threshold":"confidence of recognizers",
    "buses":[
      {
        "id":"device over activating recognition",
        "stream":"stream to be recognized",
        "bus":"bus for writing recognized gestures",
        "ip":"address of the output bus",
        "port":"port number"
      }
    ]
  }
}

```

Box 6: Configuration file for the proposed middleware.

TABLE 2: Requirements for the supported devices and the recognition algorithm.

Device	Stream	Resolution @30 fps	R/W (Mbps)	Recognition (MIPS)	Hardware requirements
Kinect 1	RGB	640 × 480 pixels	221.2	NA	Windows 7 USB 2.0 32-bit CPU DX 9.0c video card
	Depth	320 × 240 pixels	18.4	NA	
	Skeleton	20 joints	0.633	0.125	
Kinect 2	RGB	1920 × 1080 pixels	1492.9	NA	Windows 8.1 USB 3.0 64-bit CPU DX 11 video card
	Depth	512 × 424 pixels	52.1	NA	
	Skeleton	26 joints	0.824	0.160	
RealSense F200	RGB	1920 × 1080 pixels	1492.9	NA	Windows 10 64-bit CPU
	Depth	640 × 480 pixels	73.7	NA	
	Skeleton	22 joints	0.697	0.136	

As regards recognition, we have computed the time complexity required for recognizing gestures by using Hidden Markov Models [33], in the hypothesis that observed and hidden variables are discrete. We will assume that the recognizer is able to classify  $G$  different gestures, which means that  $G$  different Hidden Markov Models will be trained. Also, we assume recognition is *continuous*, which means any time a new frame is read from the input device, and then recognition is redone. In order to compute the exact number of needed operations, let us define the following quantities:

- (i)  $G$  represents how many gestures can be recognized.
- (ii)  $J$  represents the number of components of one frame in the gesture.
- (iii)  $F$  represents the frame rate of the input device.
- (iv)  $Z$  represents the number of clusters containing the gesture space, that is, the number of observable states.
- (v)  $K$  represents the number of hidden states.
- (vi)  $T$  represents the time length of a gesture.

Basically, the tasks needed to classify a gesture are two: (i) discretization of the input gesture by means of clustering and (ii) computation of the probability that the observed gesture belongs to each of the trained HMMs by using the so-called *Forward-Backward* algorithm [34].

First of all, let us assume the input space is partitioned into  $Z$  equal hypercubes (i.e., clusters). The former task consists in the computation of the centroid closest to the input gesture frame, that is, computing the Euclidean distance between the frame and the  $Z$  centroids.

The latter one is a recursive technique called *Forward-Backward*, applied  $T$  times, which computes the probability (represented as a  $K \times K$  matrix) of having the observed gesture at time  $t \in \{0, 1, \dots, T\}$ . The technique is applied to each of the  $G$  HMMs and the one with the highest probability will represent the recognized gesture.

The most complex and predominant operation of the two above tasks is the floating point multiplication. Without loss of generality, we assume here that one floating point multiplication consists in a single CPU instruction. The total number of instructions per second  $IPS_{\text{tot}}$  required for recognition is

$$IPS_{\text{tot}} = GF \left( \overbrace{\frac{Z}{J}}^{\text{Cluster selection}} + \overbrace{\frac{TK^2}{T}}^{\text{Forward algorithm}} \right). \quad (1)$$

For reader's commodity, we reported in the table the normalized version of  $IPS_{\text{tot}}$ , that is,  $IPS_{\text{tot}}/G$ , which represents the number of operations needed by the recognizer for computing the output probability from just *one* Hidden Markov Model. In the real-case scenario, such a value must be clearly multiplied by  $G$ . Also, as explained in [35], some typical values for all of the above parameters are  $F = 30$ ,  $Z = 64$ ,  $K = 4$ , and  $T = 20$ ; the value of  $J$ , in a 3D space, is simply three times the number of tracked joints.

**4.2. Experimental Assessment.** In this section we will report the timing performance of our middleware for two typical real-world applications: (i) multiple clients requiring accessing the same devices channel; (ii) multiple recognition algorithms running simultaneously for different connected devices.

In our experiments we used HP Laptop Model 7265NGW (year 2016) mounting an Intel i7-6500 at 2.6 GHz, 16 GB of RAM, 4 USB 3.0 ports, 1TB Crucial CT105 SSD with Windows 10 at 64 bits. We connected a Kinect version 1, a Kinect version 2, and a RealSense F200 camera and then run our middleware. In all the experiments the devices were set to 30 fps.

**Bandwidth Demanding Test.** The first experiment was aimed at checking the performance of the middleware for a typical highly bandwidth demanding scenario.

In order to do so, we connected the input devices and accessed their skeleton channels from a variable number of connected clients (WebSockets) running on the same machine. Then we checked the average latency between the time a frame was read from the input device and the time the same frame was written to the related output bus.

Figure 4 reports the observed latency by varying the number of clients connected to one or more of the input devices. The read-write (R/W) latency increases almost linearly with the number of connected clients. This is due to the WebSocket bus (see Section 3.2) which manages only unicast connections. As suggested by [36, 37], we fixed a latency threshold to 0.1 seconds, as it is the maximum acceptable delay in a real time visualization task. We thus obtained that the middleware can support up to 105 connected clients before the latency becomes unacceptable. A possible improvement may be the implementation of a *multicast* bus which may allow for much more connected clients.

**CPU Demanding Test.** The second experiment was aimed at verifying the performance in the case of a highly CPU demanding task such as gesture recognition.

First, we have recorded into our MongoDB database four different gestures (represented as skeleton sequences), namely, the *swipe left to right*, *swipe right to left*, *zoom-in*, and *zoom-out* from 24 different users. Each gesture was repeated two times by the users which stand in front of the Kinect v1, Kinect v2, and RealSense F200.

After that, we trained one Hidden Markov Model for each gesture (and for each different device) by using the previously recorded gestures as training set.

In the testing stage we attached the input devices to the middleware while one user was placed in front of them to perform some of the aforementioned gestures. Then we have run the gesture recognition service and checked the latency between the time input skeleton frame was read from the device and the time the gesture was classified. In order to check the CPU pressure, we varied the length of the windowed buffer containing the last  $T$  observed frames to be fed in input to the HMM.

The first three rows of Figure 5 report the recognition latency in the case of one input device performing recognition

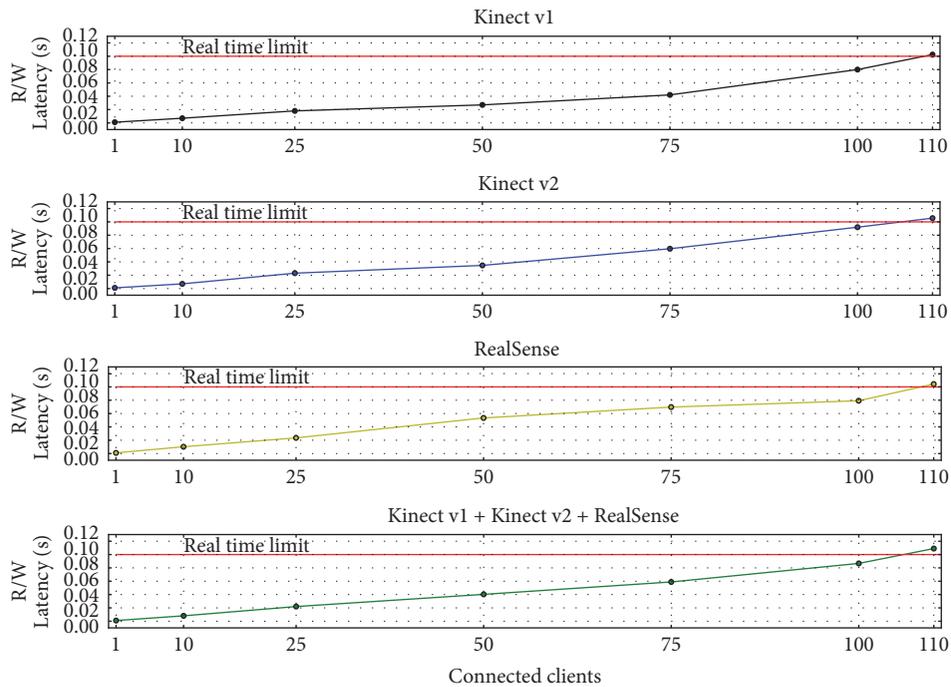


FIGURE 4: Read/write latency of skeleton frames by varying the number of connected clients.

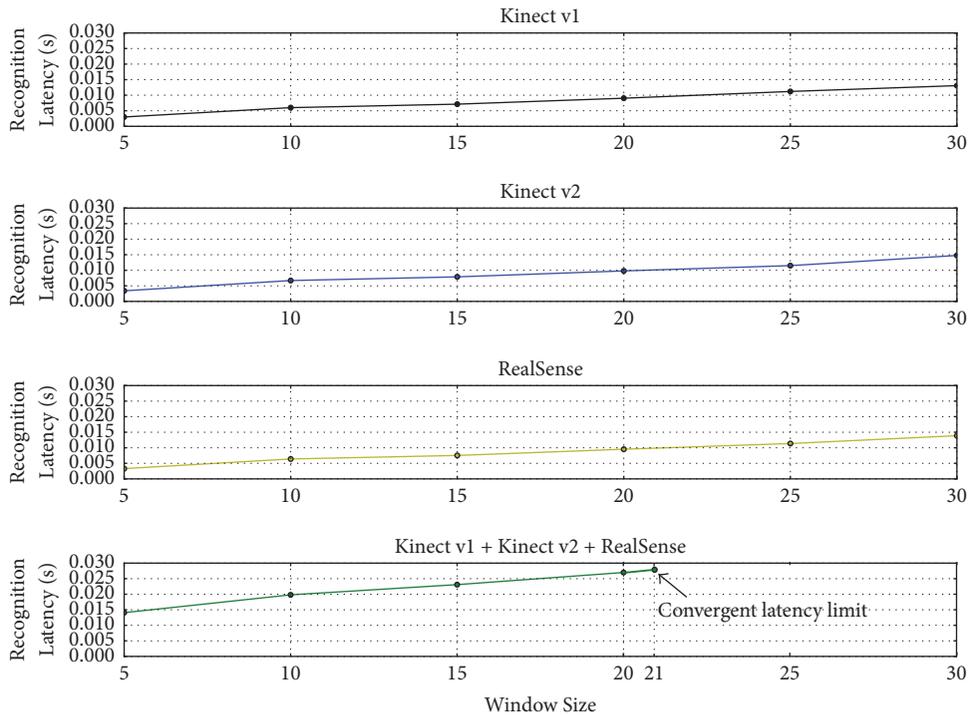


FIGURE 5: Recognition latency for different connected device.

by varying the length of the window buffer. As it can be easily seen, the latency varies according to a linear dependency (the result is in line with the result of (1)) and appears to be independent of the input device.

Finally, the fourth row of the figure reports the recognition latency in the case of three attached input devices performing recognition simultaneously. Still the dependency is linear, but for values of the window buffer greater than 21

we obtained a divergent latency (i.e., the recognition becomes computationally too expensive for the middleware). This can be explained as follows: the middleware runs *continuous* gesture recognition which means that the *Forward-Backward* algorithm is performed every time a frame is read from the input device. Thus, provided that  $F$  is the frame rate of the input device, then, the middleware has at most  $1/F$  seconds for gesture recognition. On the contrary, the latency of

recognition becomes growing, and the calls to the recognition method would rapidly fill up the program stack. In our test, since we fixed  $F = 30$  the maximum feasible latency resulting is equal to 0.033 seconds, as confirmed by the breakpoint in the figure.

There are two ways to overcome this limitation: down-sampling the input stream (but this may result in poor recognition accuracy) or implementing an alternative version of the continuous gesture recognition, named *isolated gesture recognition* [38] which runs recognition only after having identified the boundary frames of a gesture.

## 5. Conclusion and Future Works

In this paper we have described a modular middleware which aims at making the development of gesture interaction applications easy. In particular, the middleware provides some basic communication features to access gestural input devices such as the Microsoft Kinect or the RealSense cameras. All the middleware functionalities are provided by REST-based web services.

We have conducted a study on the performance of the middleware aimed at discovering its hardware and software limitations. First, the study reports the exact amount of read/write bandwidth for each of the currently supported device streams, as well as the required amount of CPU instructions in a typical task of gesture recognition; this data can be used by the middleware end-users which can easily check the requirements of their own applications.

We have also tested the middleware performance in a real-case scenario in order to check (i) how many clients can simultaneously read a single device stream with a real time constraint and (ii) how many recognition tasks can be run in parallel before the CPU load becomes unmanageable for the middleware.

The obtained results allow room for future improvements. First of all, we are planning to add support for *multicast* connections which should allow much more simultaneously connected clients to read different device streams. Also we are working on the implementation of *isolated gesture recognition*, which is much less computationally expensive than its continuous counterpart. This would allow for more simultaneous recognition tasks to be performed in real-time.

Finally, we are also working to include support to other classes of input devices (such as the Leap Motion Controller) as well as to new gesture recognition algorithms, for example, the Dynamic Time Warping or Support Vector Machines in order to allow users to choose the more suitable recognition algorithm for their own gesture datasets.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work is funded by a research grant by the Italian Ministry of University and Research, namely, project NEPTIS (Grant no. PON03PE\_00214\_3).

## References

- [1] M. L. Knapp, J. A. Hall, and T. G. Horgan, *Nonverbal Communication in Human Interaction*, Cengage Learning, 2013.
- [2] D. Wigdor and D. Wixon, *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*, Morgan Kaufmann Publishers Inc, 2011.
- [3] V. Gentile, S. Sorce, A. Malizia, D. Pirrello, and A. Gentile, "Touchless interfaces for public displays: can we deliver interface designers from introducing artificial push button gestures?" in *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI '16)*, pp. 40–43, ACM, Bari, Italy, June 2016.
- [4] D.-D. Dinh, J. T. Kim, and T.-S. Kim, "Hand gesture recognition and interface via a depth imaging sensor for smart home appliances," *Energy Procedia*, vol. 62, pp. 576–582, 2014.
- [5] E. Daidone and F. Milazzo, "Short-term sensory data prediction in ambient intelligence scenarios," in *Advances onto the Internet of Things*, pp. 89–103, Springer, 2014.
- [6] F. Sadri, "Ambient intelligence: a survey," *ACM Computing Surveys*, vol. 43, no. 4, article 36, 2011.
- [7] A. De Paola, G. L. Re, F. Milazzo, and M. Ortolani, "Adaptable data models for scalable ambient intelligence scenarios," in *Proceedings of the International Conference on Information Networking (ICOIN '11)*, pp. 80–85, January 2011.
- [8] A. Škraba, A. Koložvari, D. Kofjač, and R. Stojanovič, "Wheelchair maneuvering using leap motion controller and cloud based speech control: prototype realization," in *Proceedings of the 4th Mediterranean Conference on Embedded Computing (MECO '15)*, pp. 391–394, IEEE, Budva, Montenegro, June 2015.
- [9] V. Gentile, A. Malizia, S. Sorce, and A. Gentile, "Designing touchless gestural interactions for public displays in-the-wild," in *Proceedings of the 17th International Conference on Human-Computer Interaction*, pp. 24–34, Springer, Los Angeles, Calif, USA, 2015.
- [10] V. Gentile, S. Sorce, A. Malizia, and A. Gentile, "Gesture recognition using low-cost devices: techniques, applications, perspectives," *Mondo Digitale*, vol. 15, no. 63, pp. 161–169, 2016.
- [11] S. Fong, J. Liang, I. Fister, and S. Mohammed, "Gesture recognition from data streams of human motion sensor using accelerated PSO swarm search feature selection algorithm," *Journal of Sensors*, vol. 2015, Article ID 205707, 16 pages, 2015.
- [12] P. Ramanahally, S. Gilbert, T. Niedzielski, D. Velázquez, and C. Anagnost, "Sparsh UI: a multi-touch framework for collaboration and modular gesture recognition," in *Proceedings of the ASME/AFM World Conference on Innovative Virtual Reality (WINVR '09)*, pp. 137–142, February 2009.
- [13] U. Laufs, C. Ruff, and J. Zibuschka, "Mt4j-a cross-platform multi-touch development framework," in *Proceedings of the Workshop of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, Berlin, Germany, June 2010.
- [14] B. Signer, U. Kurmann, and M. C. Norrie, "iGesture: a general gesture recognition framework," in *Proceedings of the 9th International Conference on Document Analysis and Recognition (ICDAR '07)*, vol. 2, pp. 954–958, September 2007.
- [15] J. Bas, *A 3D gesture recognition extension for iGesture [Ph.D. thesis]*, Vrije Universiteit Brussel, 2011.
- [16] M. Muller, "Dynamic time warping," in *Information Retrieval for Music and Motion*, pp. 69–84, Springer, Berlin, Germany, 2007.

- [17] J. Roltgen, *AQUA-G: a universal gesture recognition framework [Ph.D. thesis]*, Digital Repository Iowa State University, 2010.
- [18] S. Carrino, E. Mugellini, O. Abou Khaled, and R. Ingold, "ARAMIS: toward a hybrid approach for human-environment interaction," in *Human-Computer Interaction. Towards Mobile and Intelligent Interaction Environments: 14th International Conference, HCI International 2011, Orlando, FL, USA, July 9–14, 2011, Proceedings, Part III*, vol. 6763 of *Lecture Notes in Computer Science*, pp. 165–174, Springer, Berlin, Germany, 2011.
- [19] S. Ruffieux, D. Lalanne, E. Mugellini, and O. A. Khaled, "Gesture recognition corpora and tools: a scripted ground truthing method," *Computer Vision and Image Understanding*, vol. 131, pp. 72–87, 2015.
- [20] Z. Chaczko, C. Y. Chan, L. Carrion, and W. M. G. Alenazy, "Haptic middleware based software architecture for smart learning," in *Proceedings of the Asia-Pacific Conference on Computer-Aided System Engineering (APCASE '15)*, pp. 257–263, Quito, Ecuador, July 2015.
- [21] V. Gentile, S. Sorce, and A. Gentile, "Continuous hand openness detection using a kinect-like device," in *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '14)*, pp. 553–557, July 2014.
- [22] R. Perrey and M. Lycett, "Service-oriented architecture," in *Proceedings of the Symposium on Applications and the Internet Workshops (SAINT '03)*, pp. 116–119, January 2003.
- [23] F. Wang, L. Hu, J. Zhou, and K. Zhao, "A data processing middleware based on SOA for the internet of things," *Journal of Sensors*, vol. 2015, Article ID 827045, 2015.
- [24] F. Moreno, E. Ramírez, F. Sans, and R. Carmona, "An open source framework to manage kinect on the web," in *Proceedings of the 41st Latin American Computing Conference (CLEI '15)*, pp. 1–9, Arequipa, Peru, October 2015.
- [25] M. Quigley, K. Conley, B. Gerkey et al., "ROS: an open-source robot operating system," in *Proceedings of the ICRA Workshop on Open Source Software*, Kobe Japan, 2009.
- [26] Sensors-ROS wiki, <http://wiki.ros.org/Sensors>.
- [27] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education India, 1995.
- [28] <https://www.mongodb.com/it>.
- [29] D. Crockford, "The application/json media type for javascript object notation (json)," Tech. Rep., RFC Editor, 2006.
- [30] "Kinect v1 documentation," <https://msdn.microsoft.com/en-us/library/hh855359.aspx>.
- [31] Kinect v1 documentation, <https://developer.microsoft.com/en-us/windows/kinect/hardware>.
- [32] Kinect v1 documentation, <https://communities.intel.com/docs/DOC-24012>.
- [33] How to do gesture recognition with kinect using hidden Markov models, <http://www.creativedistracted.com/demos/gesture-recognition-kinect-with-hidden-markov-models-hmms/>.
- [34] C. M. Bishop, *Pattern Recognition and Machine Learning*, Information Science and Statistics, Springer, New York, NY, USA, 2006.
- [35] F.-S. Chen, C.-M. Fu, and C.-L. Huang, "Hand gesture recognition using a real-time tracking method and hidden Markov models," *Image and Vision Computing*, vol. 21, no. 8, pp. 745–758, 2003.
- [36] B. A. Myers, "Importance of percent-done progress indicators for computer-human interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '85)*, pp. 11–17, San Francisco, Calif, USA, April 1985.
- [37] J. Nielsen, *Usability Engineering*, Elsevier, 1994.
- [38] Y. Wu and T. S. Huang, "Vision-based gesture recognition: a review," in *Proceedings of the International Gesture Workshop*, pp. 103–115, 1999.



**Hindawi**

Submit your manuscripts at  
<https://www.hindawi.com>

