Contents

1 Introduction **1**

2 Background **4**

2.1 Related Technologies and Frameworks 4

2.1.1 Single Display Groupware 4

2.1.2 Frameworks and Toolkits 8

2.1.3 Multi-modal Fusion Engines 9

2.2 Use of Gestures in Applications and Consumer Electronics 16

2.3 iGesture Basics 20

2.3.1 Comparison with Multi-modal Fusion Engines 25

3 A Multi-modal Gesture Recogniser **26**

3.1 Multi-modal Recognition Architecture 26

3.2 Composite Gesture Definition 29

3.3 Multi-modal Recognition Algorithm 30

4 Implementation **34**

4.1 Device Manager 34

4.1.1 Functionality 35

4.1.2 Implementation 36

4.2 Integration of New Gesture Devices 40

4.2.1 Integration of the Wii Remote in iGesture 40

4.2.2 Integration of TUIO in iGesture 45

4.2.3 Changes to the iGesture Workbench GUI 49

4.3 Multi-modal Composite Gestures 50

4.3.1 XML Schema 50

4.3.2 Composite Descriptor GUI 55

4.4 Implementation of a Multi-modal Recogniser 56

4.4.1 Multi-modal Recognition Architecture Components 56

4.4.2 Multi-modal Recognition Algorithm 57

4.4.3 Constraints 58

4.4.4 Composite Test Bench 63

5 Applications **66**

5.1 Multimedia Player 66

5.2 Presentation Tool 68

5.3 Geco 69

6 Summary and Future Work **70**

A UML Diagrams **72**
A.1 Device Manager 73
A.2 TUIO 78
A.3 Multi-modal Gestures 83

B XML Schema **86**
B.1 iGestureSet.xsd 87
B.2 descriptor.xsd 89
B.3 constraint.xsd 92

Bibliography **101**

Abstract

Nowadays, more and more commercial products support gesture based interaction. Some of the best known examples are Nintendo's Wii gaming console and Apple's iPhone. The commercial success of these products validates the usefulness of gesture interaction. Firstly, it makes devices and software easier to use by providing more natural interfaces. Secondly, they support and attract a broader range of users. Audiences that normally do not frequently play games, such as women or adults, represent a steadily increasing audience since the introduction of gesture-controlled devices. However, gesture interaction can not ony be used for gaming but also gains popularity in desktop computing.

Unfortunately, it is still difficult to develop applications that support gesture interaction. Existing frameworks to build these types of applications either offer a limited and fixed number of gestures or provide limited support for algorithm and gesture designers. In many cases, the gesture interaction functionality is hard-coded in specific applications, resulting in a more cumbersome and complex development process.

iGesture is an open source, Java-based framework for gesture recognition that provides support for the application developer as well as algorithm and gesture designers. New devices and recognition algorithms can easily be added. Gestures can be grouped and managed in gesture sets and new gestures can be defined by sample or textual descriptors.

In this thesis, we have extended the iGesture framework with support for composite gestures and multi-modal interaction. Composite gestures allow the gesture designer to define complex gestures based on a set of simpler gestures in a declarative manner. A small set of basic gestures may lead to better recognition rates since the gestures forming part of this set are more distinctive. Multi-modal interaction makes it possible to, for example, combine gesture input with voice input and thereby supports the invocation of actions in a more natural way.

To achieve the goal of declarative gesture composition, a multi-modal recogniser has been developed to recognise composite and multi-modal gestures. Furthermore, we also defined an extensible set of composite gesture constraints. Last but not least, tool support for designing and testing composite gestures is provided as well.

Acknowledgements

Gesture and multi-modal interaction are two very interesting topics. More and more hardware and software products provide support for gesture and/or multi-modal interaction. This thesis provided me the opportunity to do some research and acquire knowledge in this specific domain of computer science.

At the same time, I was able to contribute to the iGesture open source project. As a recent Linux and open source software user, this was my first opportunity to give something back to the community.

Hereby, I would like to acknowledge everyone who helped me during the realisation of this project. First of all, I would like to thank my parents for giving me the opportunity to obtain this Master's degree and supporting me throughout my studies at the Vrije Universiteit Brussel.

I would also like to thank my promoter, Prof. Dr. Beat Signer. He was always prepared to answer my questions and give me advise when I needed it the most. I would also like to thank him for proofreading my thesis and suggesting improvements.

I would also like to thank Ueli Kurmann, the founder of the iGesture framework, for his help and advise regarding the iGesture framework.

Finally, I would like to thank Ruben Dequeker for proofreading my thesis.

Chapter 1

Introduction

With the introduction of the Wii gaming console, Nintendo was the first company to introduce gesture interaction in mainstream consumer products. Nintendo was a real trend-setter and soon after the introduction of the Wii all of Nintendo's competitors in the gaming console market came with similar solutions. Other markets such as the computer hardware and software industry followed the trend as well and introduced gesture interaction support in their products.

Gesture interaction and more generally multi-modal interaction allow the user to interact with a gaming console or with a computer in a more natural way. Multi-modal interaction combines multiple modalities. Some examples of input modalities are pen, touch and voice input, 3D gesture devices, eye tracking, facial expressions and body location. Text, sound and speech are examples of output modalities. Interacting multi-modally with a computer offers several advantages. The most important advantages are:

- a greater task efficiency, expressive power and precision
- better error recovery
- support for a broader range of users.

Some examples are used to demonstrate how multi-modal and gesture interaction help in facilitating specific tasks. A first example is drawing an UML diagram or any other kind of diagram. Drawing an UML diagram on paper is fast and straightforward. For a class, the user draws a rectangle and a folder for a package. Drawing the same diagram with a diagram editor may be cumbersome. The user has to select the correct shape from the palette and drag it to the canvas to add it to the diagram. Figure 1.1 shows an UML diagram editor with the palette on the left-hand side and the canvas on the right-hand side. The user selects the `class` item in the palette and drags a `class` object on the canvas.

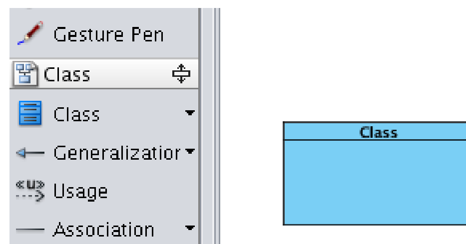
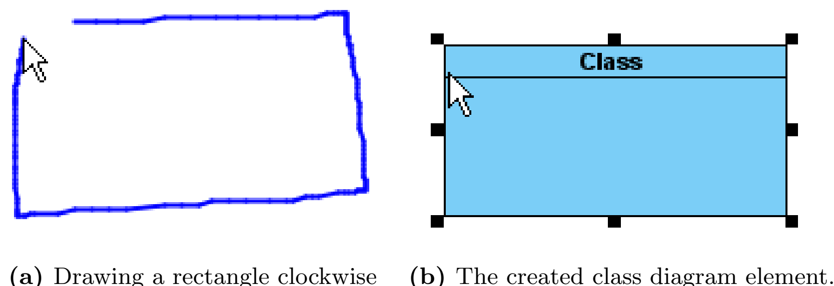


Figure 1.1: Palette and canvas in a classical diagram editor

Gestures can be used to make this task more natural. The user draws a rectangle or a folder shape with the mouse on the canvas and the diagram editor replaces it by the corresponding diagram element as shown in Figure 1.2.



(a) Drawing a rectangle clockwise **(b)** The created class diagram element.

Figure 1.2: Drawing a class object using gestures

More and more browsers provide support for gestures as well. For example, by holding the right mouse button and dragging the mouse to the left or to the right, the user can browse back and forth through the browser's history. This behaviour can be compared to the flicking through the pages of a book or newspaper.

Another example is a digital city guide which enables mobile users to access restaurant and subway information. If the user wants to know the location of an Italian restaurant, they can give the voice command "Italian restaurants near the Empire State building". The user can also combine voice with a touch gesture by giving the command "Italian restaurants near here" and touching the location of the Empire State building on the map. Another possibility is to circle the area around the Empire State building on the map and write the word "Italian" to get the same search results.

Several frameworks have been developed to simplify the implementation of applications that support gesture and/or multi-modal interaction. However, these frameworks may have a limited gesture set and limited support to design and add new gestures. In other frameworks, the gestures may be hard-coded. A framework is needed that supports the application developer as well as the (gesture) interaction and recognition algorithm designers.

That is where iGesture comes into play. iGesture is an open source, Java-based framework for gesture recognition. It provides support for different gesture devices such as digital pens, mice and graphics tablets. Multiple gesture recognition algorithms are supported. iGesture also provides support to add new devices and to add and test new recognition algorithms. The iGesture Workbench enables the gesture designer to design, create and manage gestures.

The purpose of this thesis was to extend the iGesture framework with functionality to combine gestures and enable multi-modal interactions. A client application could combine simple gestures without any iGesture support. However, this introduces extra complexity in the development of an application, leads to hard-coded solutions and might introduce delays in the recognition process. Therefore, specific functionality has been added to iGesture in the form of a multi-modal recogniser to support application developers with the recognition of composite and multi-modal gestures. An extensible, predefined set of composite gesture constraints (e.g. a sequence of gestures, concurrent gestures, time- and proximity-based constraints) and a test tool have been developed to support gesture designers in designing and testing the recognition rate of composite and multi-modal gestures.

We start in Chapter 2 by discussing the advantages of multi-modal and gesture interaction as well as investigating related work. The chapter continues with numerous examples of applications,

frameworks and devices that support gesture interaction. An introduction to the basics of the iGesture framework concludes Chapter 2. Chapter 3 introduces the general concepts related to the multi-modal recogniser such as the multi-modal recognition architecture, the recognition algorithm and the predefined constraints. The implementation of the multi-modal recogniser and support for new gesture devices—the Wii Remote and TUIO devices—is further elaborated in Chapter 4. Different examples of applications that can benefit from multi-modal gesture support are given in Chapter 5 and some conclusions as well as a discussion about future work are provided in Chapter 6.

Chapter 2

Background

In the first section of this chapter, several related technologies and frameworks are discussed. The second section looks at some examples of applications and consumer electronics that incorporate support for gestures. The chapter concludes with a description of the iGesture framework which constitutes the basis that is needed to understand the rest of the thesis.

2.1 Related Technologies and Frameworks

The goal of this thesis was to extend the iGesture framework with multi-modal gesture functionality. Multi-modal gestures are performed with multiple input devices by one or more users. In literature, several frameworks, toolkits and technologies that try to achieve similar goals can be found. We start this section with a discussion of Single Display Groupware, followed by some example frameworks and toolkits. Finally, at the end of this section Multi-modal Fusion Engines are introduced.

2.1.1 Single Display Groupware

Multi-modal and composite gestures can be of great use in applications where users work together. Computer Supported Collaborative Work (CSCW) [33] is the domain in computer science that deals with user collaboration supported by computers. The users can be located at the same place (co-located collaboration) or at different places (remote collaboration). In both cases users can collaborate at the same time or at a different time. Figure 2.1 shows the different types of computer supported collaboration with some examples and related technologies.

We are mainly interested in users collaborating at the same time and location with a single system. Single Display Groupware (SDG) is an example of a technology that supports this kind of collaboration. Stewart, Bederson et al. [5, 28] define Single Display Groupware as computer programs which enable co-present users to collaborate via a shared computer with a single shared display and simultaneously use multiple input devices. Consequently, Single Display Groupware is also known under the name co-located or co-present collaboration groupware.

The current operating systems are designed with single user interaction in mind and their design makes co-present collaboration difficult. First of all, keyboard and mouse are not independent input devices. The mouse is used to determine the keyboard focus. Secondly, if multiple pointing devices are present on a computer, they share the same input channel. Either they share the system cursor or only one pointing device can be active at a time. Another issue affects the widgets used in current user interfaces. Most of the widgets are not suited for concurrent use. For example, what should

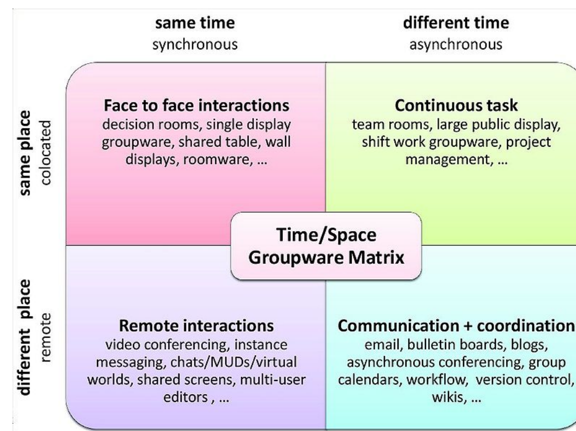


Figure 2.1: CSCW Matrix

happen if multiple users interact with different parts of a scrollbar or a menubar or if they select the same object?

Single Display Groupware tries to solve these issues by, for example, providing multiple cursors. The widgets can be either one-at-a-time or anyone-mixed-together widgets [5]. One-at-a-time widgets are only accessible by one user at a time like a dropdown menu. A canvas is an example of a anyone-mixed-together widget since multiple users can draw on the canvas without any problem. In some cases social protocols are sufficient to prevent conflict. Another alternative is to avoid traditional widgets like pull-down menus and replace them with 'local tools' [5]. The user can pick up the tool by clicking on it and can then use it. Each tool has its own state, so there is no need to keep a global state or a state for each user.

In [5, 28] Stewart et al. performed a user study with the KidPad application. KidPad is a painting program where multiple children can draw at the same time and cooperate on the same drawing or help each other. The results of their study showed that existing single user technology leads to conflicts in a co-present collaborative setting. The reason is that the input device has to be shared and this leads to an unequal level of control. Often the user that is in charge is not always the user that handles the input device.

The user study also shows that SDG is well suited for co-present collaboration but the efficiency still heavily depends on human characteristics and personalities as well as the context of use. Over the years, several frameworks and toolkits have been developed to facilitate the creation of SDG applications. In the following, MIDDesktop and SDG Toolkit are discussed as two examples of SDG frameworks for SDG.

MIDDesktop

MIDDesktop [25] is a Java framework that is multi-mouse aware. Within this application shown in Figure 2.2, regular Java applets can be run in separate windows. MIDDesktop itself interprets multiple mouse inputs. This approach allows applets to be used in single user mode but also by multiple users at once, without any extra programming effort for the developer. Unfortenately, the framework was only supported on Windows 98 and has been discontinued afterwards.

MIDDesktop tries to make the development of applications that support multiple input devices and multiple users as easy as possible. It achieves this goal by hiding the details and by managing different users and input devices. In that respect, iGesture is very similar to MIDDesktop since it

also handles different users and input devices in a flexible way. iGesture can be used to create Java applets and applications while MIDDesktop only supports applets.

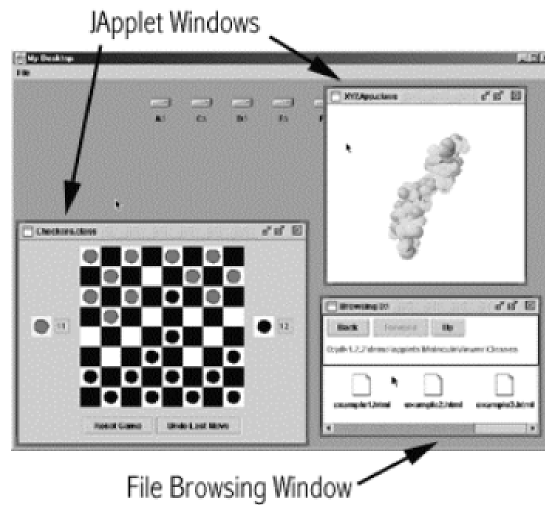


Figure 2.2: MIDDesktop

SDG Toolkit

The SDG Toolkit [29, 30] focusses on two main goals. Firstly, it allows multiple users to concurrently use their mouse in a single system. The second goal is to provide user interface (UI) components that can deal with concurrent use.

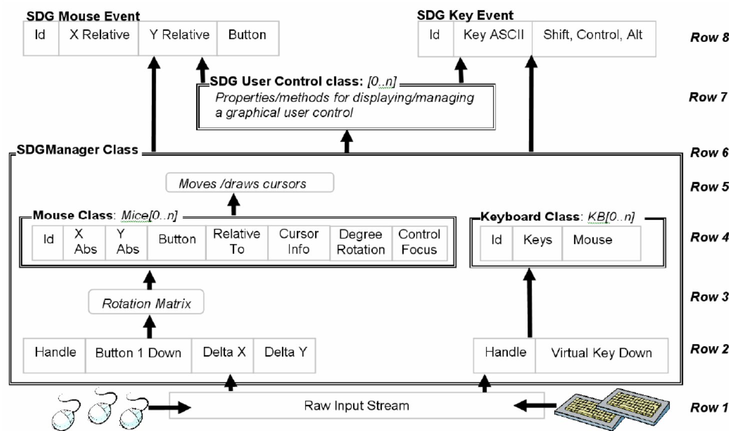


Figure 2.3: SDG Toolkit

Figure 2.3 shows how the SDG Toolkit[1] works and addresses the issues mentioned before. The first problem is the fact that all windowing systems combine the input of multiple mice and keyboards into a single system mouse and system keyboard input stream. Only this single input stream is available to a programmer. For non-standard input devices, either low-level code (e.g. a device driver) must be written or an API has to be used, which do not always work well together with the windowing system.

[1] http://grouplab.cpsc.ucalgary.ca/cookbook/index.php/Toolkits/SDGToolkit

The SDG Toolkit uses Microsoft's raw input model which allows the programmer to get a list of all attached input devices and to parse the input stream to identify which input device created the event or input. The `SDGManager` class handles the input devices and parses the raw input streams. Note that Raw Input is however a technology that is only available under Windows XP. The `SDG Mouse Event` and `SDG Key Event` classes add an ID field to the standard mouse and key events. In this way, the device that generated an event can be identified.

Another issue is the single system cursor. Multiple cursors can be displayed by using top-level transparent windows with a cursor drawn on them. These transparent windows are moved by the `SDGManager` after each mouse move. A text label can further be added to distinguish between the different cursors.

The combination of the system mouse and SDG mice has its side effects. Since the window system merges multiple pointer inputs to move a single system cursor, this cursor is still moving around on the screen among the SDG mice. The behaviour of this mouse is unpredictable since it reacts on the combined forces of the SDG mice. Making the cursor invisible does not solve the problem. A click generated by an SDG mouse generates a click on the system mouse and at the same time activates any window or widget under the system mouse. A possible solution is to position the system mouse at an unused area of the UI. The downside of this approach is that the user cannot use standard widgets or window controls (e.g. resize, close) anymore and also not switch to non-SDG windows. Another solution is to bind the system mouse to one of the SDG mice, changing the SDG mouse to a super mouse. A side effect of this solution is that if the user with the super mouse clicks outside of the SDG application, in a non-SDG window, the SDG application loses focus. As a consequence, the other SDG mice will not respond anymore. Either solution has its advantages and disadvantages and both have been implemented in the SDG Toolkit. To handle multiple keyboard foci, every keyboard is associated with a mouse. Any keyboard input is then directed to the control the associated mouse is focused on.

Mouse events and cursor appearance assume a single orientation, but if the users are sitting around a (touch) table, the users may have a different orientation. This would make the system not very usable since the mouse seems to move in the wrong direction. To solve this problem, for every mouse the orientation can be defined in the `SDGManager`. The mouse coordinates and orientation are then adjusted with the corresponding rotation matrix.

Conventional controls cannot distinguish between SDG users and they are not designed to handle concurrent use. Therefore, the toolkit includes a set of SDG aware controls based on their single-user variants. Also interfaces and an event mechanism is included. If controls implement such an interface, the `SDGManager` can invoke those methods and pass the arguments.

Compared to MIDDesktop, the SDG Toolkit offers solutions for conflicts that can arise when traditional UI components are concurrently used by multiple users. For example, what should happen if two users select a different value in the same combobox at the same time? Or what happens when two users want to move the same slider in a different direction? The SDG Toolkit provides components that can handle these situations.

Unlike the SDG Toolkit, iGesture does not deal with UI components nor with the more classical forms of interaction with a computer. iGesture deals with gesture interaction between multiple users but it is up to the application developer to design the desired gestures as well as to define any associated functionality.

Although both MIDDesktop and the SDG Toolkit support the use of multiple mice within an application, the application either has to be run in the framework or has to be developed with the toolkit, which seems to be a major limitation. Therefore, the SDG community would like to have support for multiple mice and cursors on the OS level. The Multi-cursor X Window Manager [32] and Multi-Pointer X (MPX) [13] are both a step into this direction by providing support for multiple cursors and mice on the level of the window manager. This enables many existing X Window applications to run without any modification and with support for multiple pointers. MPX has been merged into the X Server since version X11R7.5. X Server now not only supports multiple cursors but also multi-touch.

2.1.2 Frameworks and Toolkits

In this section, two frameworks that are similar to iGesture are discussed.

Pointer Toolkit

The Pointer Toolkit [4] is a Java framework to develop and test software that utilises multiple input devices. To support the different input devices, existing hardware APIs are used. The input of multiple input devices is managed by a unified event system. This event system is based on the Java event architecture which implies that the application has to register itself as an event listener.

The toolkit also provides two tools for aggregating events and for debugging. The aggregation tool is the central point where all hardware input is centralised and the events are recognised. The debugging tool is used to log and analyse the input. The input flow of the Pointer Toolkit is illustrated in Figure 2.4.

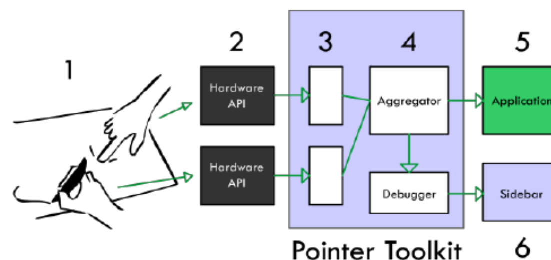


Figure 2.4: Pointer Toolkit

The architecture and goals of iGesture and the Pointer Toolkit are comparable. Both support similar devices like mice and touchtables (see 4.2.2). To provide this support existing hardware APIs are used. iGesture for example uses the Java Bluetooth API to support the Wii Remote (for more information consult Section 4.2.1). Both iGesture and the Pointer Toolkit use an event system based on the Java event architecture to notify interested parties like applications about specific input events. While the Pointer Toolkit focusses on events such as hover, drag, press and release events, iGesture enables the specification of arbitrary gestures. Last but not least, a test tool is provided in both frameworks.

Inertial Measurement Framework

The Inertial Measurement Framework [6] consists of three components: a component to capture 3D gestures, a gesture recognition algorithm and a component to combine recognised gestures. The

latter component can concurrently or consecutively combine gestures to create composite gestures. The composite gestures can then trigger output routines.

The gesture device is a wireless six-axis inertial measurement unit and contains several accelerometers and gyroscopes. The authors defined a set of atomic gestures that are characterised by the number of peaks in the accelerometer trace. A straight-line motion for example creates a two-peaked trace, while a there-and-back motion generates a three-peaked trace. Besides the number of peaks, the length and duration are also taken into account.

The recognition algorithm is executed in every dimension and the variance of the data is used to detect periods of activity. This way, the data can be filtered before it is sent to the recognition algorithm.

Composite gestures can be detected and split when two neighbouring peaks have the same polarity. Logical combinations of atomic gestures are allowed and they can be put in a temporal order. The designer only has to perform the gesture a few times and to record the recognised atomic gestures and their order. The last step is to write a script to recognise the specific combination of atomic gestures.

iGesture also provides support for 3D gestures. Instead of creating a gesture device out of existing components, there was opted to use the Wii Remote (see Section 4.2.1) or other devices. The Wii Remote is available in all consumer electronics retail stores for a decent price.

The recognition algorithm that is currently used in iGesture for 3D gestures is based on the Rubine algorithm [27, 31]. The Rubine algorithm uses a number of features that characterise the gestures instead of the number of peaks. In the 3D version of the Rubine algorithm, the standard Rubine algorithm is executed in all three planes. Using the number of peaks to recognise gestures greatly simplifies the recognition process but it limits the types of gestures that can be recognised. The features defined by Rubine enable the recognition of various kinds of gestures. Furthermore, other algorithms can easily be integrated with iGesture. Similar composite gestures can be recognised with iGesture as with the Inertial Measurement Framework.

2.1.3 Multi-modal Fusion Engines

Studies and surveys [3, 22, 11] have shown that multi-modal interfaces or interfaces that support multiple input and output modalities have several advantages. The most important advantages are

- greater task efficiency, expressive power and precision
- better error recovery
- support for a broader range of users.

Multi-modal user interfaces improve the task efficiency because the user can choose the most efficient modality to perform a task. They can combine multiple modalities or they may choose a different modality in a context-dependent manner. In a noisy environment, for example, touch input can be used instead of voice input on a mobile device. Because of the performance and efficiency advantage, users prefer multi-modal interfaces.

Multi-modal systems also facilitate error recovery for several reasons. Firstly, users select the mode they think is less error prone for that task and in doing so, they might already avoid some errors. Secondly, users use simpler language which reduces recognition errors. If an error occurs, users also tend to switch between modalities. Forth, users are less frustrated by errors when they interact with multi-modal systems. Finally, in the case that multiple modalities are combined, the semantic information for each modality can provide partial disambiguation for the other modalities.

A broader range of users is supported by the combination of several modalities. People of different age, with different skills or handicaps can use the system in a way that is most suited for them. There are a lot of different input modalities and they can be active (e.g. pen, touch and voice input, 3D gesture devices, etc.) or passive (e.g. eye tracking, facial expressions or body location). Examples of output modalities are text, sound and speech. All these modalities allow users to interact with the system in a more natural way.

Oviatt et al. [11] state several findings in cognitive psychology that explain why humans prefer to interact multimodally. Humans are able to process multiple modalities partially independent. As a consequence, presenting information with multiple modalities increases human working memory. Secondly, humans tend to reproduce interpersonal interaction patterns during multi-modal interaction with a system. Furthermore, human performance is improved when interacting in a multi-modal way because of the way human perception, communication and memory function.

The modalities influence the user interface. Multi-modal User Interfaces (MUI) differ from the traditional Graphical User Interfaces (GUI) in several aspects which are summarised in Table 2.1. While GUIs have a single event input stream, MUIs have multiple simultaneous input streams. The basic actions on a GUI (e.g. selection) are atomic and deterministic. Either the mouse position or the characters typed on a keyboard are used to control the computer. In multi-modal interfaces, the input streams are first interpreted by recognisers. This process introduces a certain degree of uncertainty. The event stream of a GUI is processed sequentially while in a MUI they have to be processed in parallel. A last difference is the used system architecture. Multi-modal systems often use a distributed architecture. For example, speech recognition is a very resource intensive process and therefore it can be beneficial to run this process on a separate server.

Table 2.1: Differences between GUI and MUI

GUI	**MUI**
Single input stream	Multiple input stream
Atomic, deterministic	Continuous, probabilistic
Sequential processing	Parallel processing
Centralised architectures	Distributed architectures

To know how a multi-modal system is constructed, we first have to understand the multi-modal man-machine interaction loop [11]. This interaction loop consists of multiple states and is shown in Figure 2.5. In the decision state, the user prepares the message they want to communicate. Then a user selects the appropriate communication channel (e.g. gesture or speech) in the action state and conveys the message.

The system is equipped with modules to capture the message. In the perception state, the system interprets the information it received from one or more sensors. Next, the system tries to give a semantic meaning to the collected information. It is also in this interpretation phase that fusion takes place. Based on the business logic and the dialogue manager rules, action is taken and an answer is generated. A fission engine determines the most appropriate output modality to transmit the answer based on the context of use and the user profile

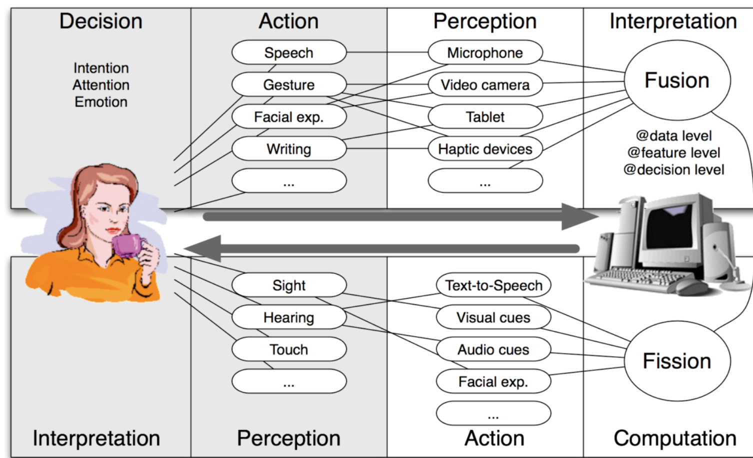


Figure 2.5: Multi-modal man machine interaction loop

From this interaction loop the main components of a multi-modal system can be derived: a *fusion engine*, a *fission module*, a *dialogue manager* and a *context manager*. The interactions between these components are illustrated in Figure 2.6.

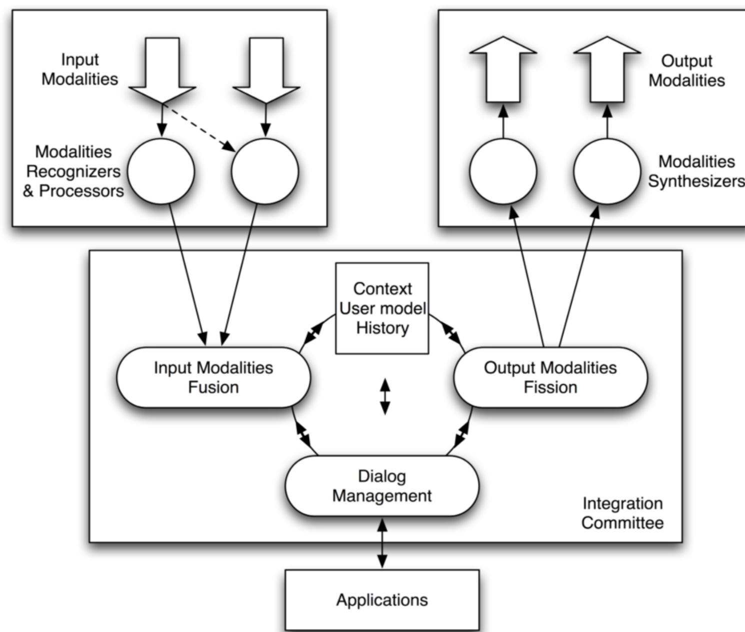


Figure 2.6: Multi-modal system architecture

The inputs are first processed by their corresponding recognisers. The recognition results are then passed to the fusion engine that fuses and interprets the inputs. Subsequently, the fusion engine sends the interpretation to the dialogue manager that will identify the actions to be taken by the applications and the message to be returned by the fission component. The latter returns the message through the most appropriate modality or modalities based on information of the context manager.

The fusion itself can be performed on three levels: the data level, the feature level or the decision level. Data-level fusion is used when the signals come from very similar modality sources (e.g. two webcams that record the same view from different angles). Since the signals are directly processed, there is no loss of information but it is susceptible to noise and more error-prone.

Feature-level fusion is applied when the modalities are tightly-coupled or time synchronised such as the fusion of speech and lip movements. It handles noise better but there is information-loss. Adaptive systems like Neural Networks (NN) and Hidden Markov Models (HMM) are frequently used to perform this kind of fusion. The downside is that those systems need a lot of training data.

Decision-level fusion is the most commonly used form of fusion since it can handle loosely-coupled modalities. It is more reliable and more accurate because of the mutual disambiguation by the semantic information of the different input modalities. An example where decision-level fusion could be applied can be found in one of the early multi-modal systems. The "Put-That-There" system was created by Bolt [7] in 1980 and combines voice with gesture interaction. The user gives the command "Put", then points to the object to move ("That") and finally points to the new location ("There").

Different types of architectures are used to perform decision-level fusion. Frame-based architectures use features to represent meaning. Features are data structures containing (attribute,value) pairs. A second type of architecture is the unification-based architecture. This architecture recursively merges (attribute,value) structures to generate a meaning representation. The last architecture is a symbolic/statistical architecture and is a kind of a hybrid system. It combines unification-based fusion with statistical processing techniques. A summary of these different types of fusion can be found in Figure 2.7.

	Data-level fusion	**Features-level fusion**	**Decision-level fusion**
Input type	Raw data of same type	Closely coupled modalities	Loosely coupled modalities
Level of information	Highest level of information detail	Moderate level of information detail	Mutual disambiguation by combining data from modes
Noise/failures sensitivity	Highly susceptible to noise or failures	Less sensitive to noise or failures	Highly resistant to noise or failures
Usage	Not really used for combining modalities	Used for fusion of particular modes	Most widely used type of fusion
Application examples	Fusion of two video streams	speech recognition from voice and lips	Pen/speech interaction

Figure 2.7: Different fusion levels and their characteristics

Fission is the process where a message is generated in the appropriate output modality or modalities based on the context and user profiles. The time synchronisation between the modalities is extremely important. If multiple commands are performed in parallel and have to be fused, the order in which they have been performed is very important because the interpretation depends on it. Oviatt et al. [11] provide the following example:

- <pointing> "Play next track": will result in playing the track following the one selected with a gesture
- "Play" <pointing> "next track": will result in first playing the manually selected track and then passing to the following track at the time "next" is pronounced
- "Play next track" <pointing>: the system should interpret the commands as being redundant.

Synchronisation is influenced by delays due to the system architecture and the used technologies (e.g. speech recognition). Therefore, distributed architectures should be used to divide the processing power.

Modelling multi-modal interaction can be very complex because of all the different input and output modalities and their combinations. User profiles and the context of use have to be taken into account as well. To simplify the modelling, two formal models have been developed: CASE and CARE.

The CASE model combines modalities at fusion engine level while the CARE model at the user level. The CASE model introduces four properties: concurrent, alternate, synergistic and exclusive. These properties describe how the modalities are combined. The combination depends on two factors. The first factor indicates whether the modalities are fused independently or combined. The second factor indicates whether the modalities are used sequentially or concurrently. Figure 2.8 shows the correlation between the different properties

		USE OF MODALITIES	
		Sequential	Parallel
FUSION OF MODALITIES	Combined	ALTERNATE	SYNERGISTIC
	Independant	EXCLUSIVE	CONCURRENT

Figure 2.8: The CASE model

The CARE model introduces the following four properties: complementarity, assignment, redundancy and equivalence. Complementarity means that multiple modalities are complementary and all of them are needed to determine the desired meaning. Bolt's Put-That-There [7] is an excellent example. Both, the voice input and the pointing gestures are needed in order to understand what the user wants to say. Assignment is used when only one modality can lead to the desired meaning (e.g. steering wheel of a car). When only one of multiple modalities is needed to understand the desired meaning, we speak of Redundancy. A user can push the play-button and at the same time speak the "play" command. Only one of the two is needed to understand what the user means. Equivalence indicates that multiple modalities are each others equivalent. They can all convey the same meaning but only one will be used at a time. For example to enter a text, the user can either use the keyboard or use voice recognition.

Example Multi-modal Systems

Several web- and non-web-based multi-modal systems have been built in the past. Web-based multi-modal systems often use W3C's EMMA markup language [2] and modality specific markup languages such as VoiceXML and InkML. EMMA or the Extensible Multi-Modal Annotation markup language provides a standardised XML representation language for encapsulating and annotating inputs for spoken and multi-modal systems. Johnston [14] presented several example applications that can be accessed from the iPhone. The applications use voice recognition (ASR), Text To Speech (TTS) and

gestures. The recognition happens on an application server. The application server consists out of an ASR server (speech recognition), a database server and a multi-modal fusion server. The client application is created by using HTML, CSS and JavaScript.

The client can send an audio stream to the ASR server which sends the recognition results back in the form of an EMMA XML document. This document is then sent to the multi-modal fusion server. The latter processes the document and returns derived results or interpretations as an EMMA XML document. The client uses the semantics to create a database query. The results from this query are then displayed to the user.

MATCH [15] or Multi-modal Access To City Help is a city guide and navigation system that enables mobile users to access restaurant and subway information for cities such as New York City and Washington D.C. The MATCH GUI either displays a list of restaurants or a dynamic map showing locations and street information. Information can be entered by using speech, by drawing on the display or by multi-modally combining the two. The user can ask for information about a restaurant such as a review, cuisine, phone number, address or subway directions to the restaurant. The results can be shown on a map and information can be synthesised to speech.

iMATCH [14] is a prototype based on MATCH for the iPhone and combines voice with touch-based gesture input. Figure 2.9 (left) shows the results of the spoken command "Italian restaurants near the Empire State building". An example of a multi-modal command is a combination of the voice command "Italian restaurants near here" and a touch gesture on the map. The diamond symbol on Figure 2.9 (right) indicates the location of the touch gesture. In the original MATCH application, the same command can be given using a stylus by circling an area and writing "italian".

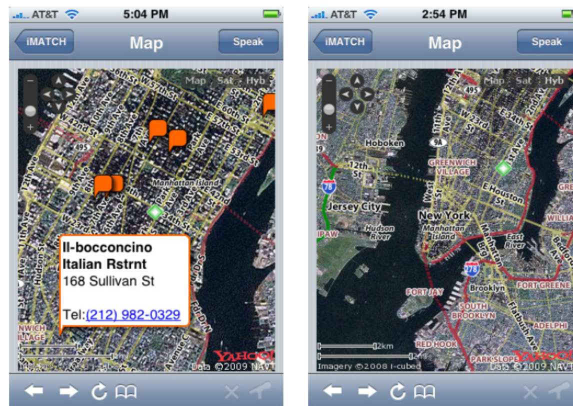


Figure 2.9: iMATCH interface

IMMIView [16] is a multi-modal system that supports co-located collaboration. It was designed to support architectural design and provides content creation and manipulation, 3D scene navigation and annotations. Laser pointers, pens, speech commands, body gestures and mobile devices are the available input modalities. The 3D scenes can be visualised on large-scale screens, head mounted displays and TabletPCs. To support these different visualisation scenarios, a user interface based on pen strokes was chosen instead of a traditional point-and-click interface. For example, to activate the main menu the user draws a triangle and the main menu appears as shown on the left in Figure 2.10. To select an option, the user draws a stroke to cross-over the option. If the user draws a lasso around an object, the object is selected and a context menu pops up.

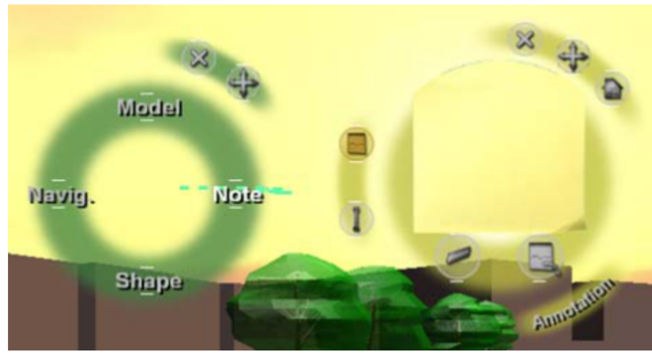


Figure 2.10: IMMIView main menu (left) and annotation menu (right)

The fusion of the different modalities is rule-based where each rule consists of preconditions and actions. If all preconditions apply, the set of actions is executed. Jota et al. performed several user evaluation tests based on ISONORM 9241-part 10 and the results show that users feel comfortable with the system and suggest that users prefer the multi-modal approach to more conventional interactions, such as mouse and menus.

ICARE [8] stands for Interaction Complementarity Assignment Redundancy Equivalence and is a component-based approach for developing multi-modal interfaces. Two kinds of components are used to build the system. Elementary components are the first kind of components. An elementary component constitutes a pure modality using device and interaction language components. The other components are composition components defining the combined usage of modalities. ICARE has been designed to make the development of multi-modal interfaces easier and has been demonstrated with a military plane simulator.

The Device component is a layer above the device driver that abstracts the raw data from the driver and enriches it by adding information like a time-stamp. An Interaction Language Component listens to a Device component and abstracts the data into commands (e.g. selection). These components rely on external descriptions for data abstraction. For example, the description of a graphical user interface is needed to abstract a selection command from raw input data.

The Composition components rely on the four properties mentioned earlier in the CARE model. Based on these properties, three composition components are defined: a Complementary, Redundancy and Redundancy/Equivalence component. Assignment and Equivalence are not modelled as components since they do not have a functional role. The Complementarity component combines complementary data that are close in time. It relies on a common structure of the manipulated objects that enables the fusion of the received pieces of data based on a set of rules. Redundancy means that two modalities contain pieces of redundant information close in time. The Redundancy/Equivalence component was introduced to simplify the specification process, and is similar to the Redundancy component except that redundancy is optional.

These components are generic because they are modality and domain independent. As many components as desired can be attached to the composition components and any composition component can be exchanged with another type, which supports the design of flexible interfaces. An example of the use and the composition of these components is shown in Figure 2.11. The Device components are located in the bottom layer, the Interaction Language components in the layer above. The next layer contains the Composition components while the top layer contains the clients.

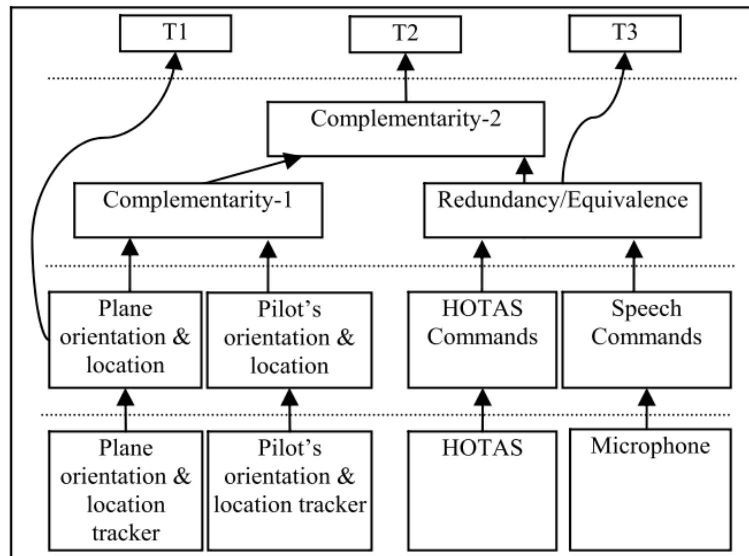


Figure 2.11: ICARE example

2.2 Use of Gestures in Applications and Consumer Electronics

Since Nintendo introduced its Wii gaming console[2] with a—at that time—revolutionary gameplay experience, a lot of other desktop applications and consumer electronics devices have as well introduced gestures as a way of control.

The Wii gaming console uses a Wii Remote to perform the gestures. A picture of the Wii Remote is shown in Figure 2.12. The revolutionary gameplay can be explained using the example of playing a tennis game. The user swings the Wii Remote like they would swing a tennis racket to hit the tennis ball. The Wii console then interprets the movement and the character in the game makes a similar movement. Every game uses movements or gestures that correspond with movements in the real world.

This revolutionary way of interacting with a game seems to be appealing to audiences that otherwise play games less often including women and adults. It seems that gesture-based control makes the games more accessible and easier to learn.

Figure 2.12: Nintendo Wii Remote (white) and PlayStation Move (black).

[2]http://www.nintendo.com/wii

Of course, other game console manufacturers do not want to miss this trend and are also working on gesture interaction solutions for their consoles. The PlayStation Move[3] is Sony's answer to Nintendo's Wii Remote. The two solutions resemble each other in a significant way as can be seen in Figure 2.12. Both devices use gyroscopes and accelerators to determine the orientation and acceleration of the remote. Note that the Wii MotionPlus extension is needed to determine the orientation. The communication with the console is in both cases via a Bluetooth connection. Even the Subcontroller resembles Nintendo's Nunchuck. However, the PlayStation's two controllers communicate wirelessly while the controllers of the Wii are connected with a cable.

The most striking difference is the orb with its size of a pingpong ball that is located at the top of the Move controller. The orb can light up in all RGB colours and is tracked by the PlayStation Eye camera. As a consequence, the exact location of the controller in 3D space can be determined. In contrast, the Wii can only determine the position relatively to its previous position. The Wii Sensor Bar can be used to determine the distance between the Sensor Bar and the Wii Remote. Therefore, the Sensor Bar emits infrared light on either end of the bar. The infrared sensor of the Wii Remote detects both infrared light sources to compute the distance via triangulation. The rotation with respect to the ground can be calculated from the relative angle of the two dots of light on the image sensor.

The use of the camera in the PlayStation Eye also supports some types of augmented reality. The user can see themselves on the screen carrying, for instance, a sword or a brush instead of the controller. Furthermore, the PlayStation Eye supports voice recognition.

On the other hand, Microsoft wants to completely remove the remote controller. Project Natal[4] will combine full body movement detection with face and voice recognition. This will take gesture interaction to the next level where *the user is the remote*. A picture of the Natal device is shown in Figure 2.13.

Figure 2.13: Microsoft Project Natal

Figure 2.14: Microsoft Surface

However, gestures are not only used to interact with gaming consoles but also to control regular desktop computers and laptops. This can either be supported on the Operating System (OS) level, via an application development framework level or on the application level.

One category of applications that uses gestures are web browsers. All major web browsers have mouse gesture support. Note that only Opera[5] offers gesture recognition as a built-in functionality,

[3]http://uk.playstation.com/games-media/news/articles/detail/item268481/PlayStation-Move-motion-controller-revealed/

[4]http://www.xbox.com/en-US/live/projectnatal/

[5]http://www.opera.com/browser/tutorials/gestures/

whereas all other web browsers use extensions to support mouse gestures. FireGestures[6] and All-in-One-Gestures[7] add mouse gestures to Firefox, Smooth Gestures[8] and Mouse Stroke[9] are examples for Chrome and last but not least there is Mouse Gestures[10] for Internet Explorer.

Firefox Mobile[11] uses drag gestures to the left and to the right to display sidebar panels. The left panel displays the available tabs, the right panel shows quick access buttons to, for example, bookmark a page.

Applications to draw graphs or diagrams can benefit from gestures as well. Visual Paradigm's UML editor[12] is an example of a gesture-enabled drawing tool. In a classical drawing tool, the palette contains all the shapes that can be drawn. By dragging a shape from the palette to the canvas, the shape is drawn. Visual Paradigm's UML editor allows the user to put shapes on the canvas by using gestures on the canvas. The user draws the gesture shown on the left in Figure 2.15 and the application replaces it with the corresponding shape displayed on the right in Figure 2.15.

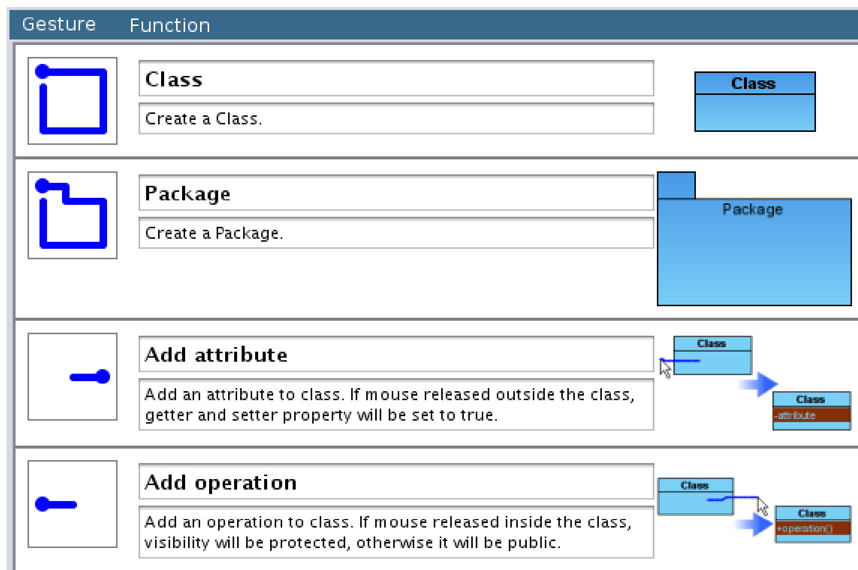


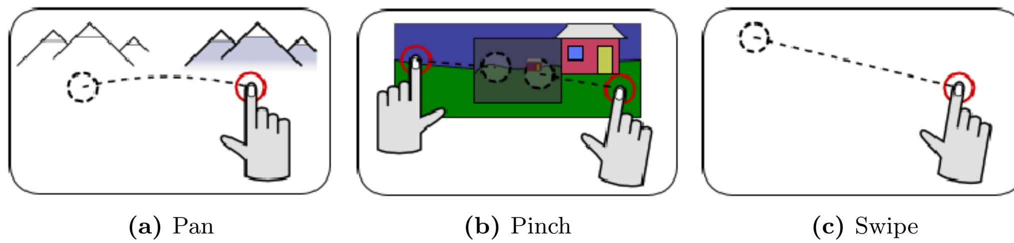
Figure 2.15: Gestures in Visual Paradigm's UML Editor

Providing support for gestures within an application development framework hides most of the implementation details and makes gestures easier to use and support in applications. Qt [21] is such a framework that is written in C++ and is available for all main operating systems. Since version 4.6, it supports multi-touch and gestures. The framework provides some default gestures but it is also possible to implement new gestures. The provided gestures are the well-known pinch, pan, swipe and tap gestures. This seems to be a good set of initial gestures since they are the most commonly used gestures to control applications.

Support for gestures and multi-touch can also be offered at OS level. Windows 7, iPhone OS and Android[13] are examples of operating systems that provide some gesture recognition functionality.

---

[6] https://addons.mozilla.org/en-US/firefox/addon/6366
[7] https://addons.mozilla.org/en-US/firefox/addon/12
[8] https://chrome.google.com/extensions/detail/lfkgmnnajiljnolcgolmmgnecgldgeld
[9] https://chrome.google.com/extensions/detail/aeaoofnhgocdbnbeljkmbjdmhbcokfdb
[10] http://www.ysgyfarnog.co.uk/utilities/mousegestures/
[11] http://www.mozilla.com/en-US/mobile/
[12] http://www.visual-paradigm.com/product/vpuml/
[13] http://googlemobile.blogspot.com/2010/03/search-your-android-phone-with-written.html



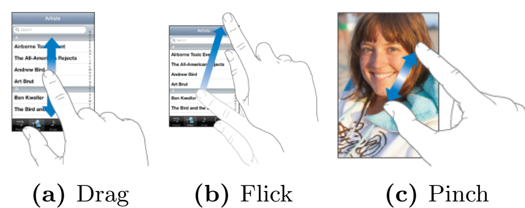
(a) Pan (b) Pinch (c) Swipe

Figure 2.16: Gestures in Qt

Moblin, Intel's Linux based operating system for netbooks, provides gesture support as well through Clutter-gesture[14]. Since version 1.7 of the X Server[15], there is also support for multiple pointers. As a consequence, multi-touch and gestures are potentially available on any Linux distribution that uses X Server version 1.7 or higher.

Besides computers and gaming consoles, other devices can benefit from gesture interaction as well. One of these devices is Microsoft's Surface [10]. The multi-touch table is shown in Figure 2.14. On the Surface table, gestures can be performed by hand or with objects.

Apple equipped all of its recent devices with multi-touch and gesture support. The iPhone[16] as well as the Magic Mouse[17] and the touchpad on the MacBook are some good examples. The iPhone can be unlocked using a swipe gesture. Tasks like scrolling and zooming can also be performed by gestures.



(a) Drag (b) Flick (c) Pinch

Figure 2.17: Gestures on the iPhone

The Magic Mouse is a mouse with a multi-touch surface. Clicking and double clicking can be done everywhere on the surface of the mouse and, for example, scrolling can be achieved through gestures. A free program called MagicPrefs[18] may be used to enable additional gestures. It is also possible to map gestures to applications which simplifies starting those applications.

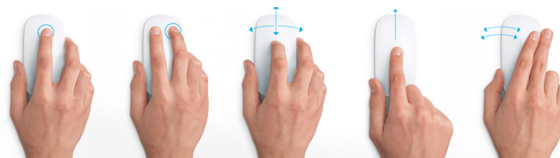


Figure 2.18: Gestures on the Magic Mouse

[14] http://moblin.org/projects/clutter-gesture
[15] http://www.x.org/wiki/Releases/7.5
[16] http://www.apple.com/iphone/
[17] http://www.apple.com/magicmouse/
[18] http://magicprefs.com/

Since Apple offers multi-touch and gesture support on the Trackpads of its MacBook product line as shown in Figure 2.19, Synaptics—the largest TouchPad manufacturer for laptops—had to introduce these features as well in order to keep their products competitive. Synaptics Gesture Suite (SGS) for TouchPads[19] provides this functionality and offers gestures such as two-finger scrolling, two-finger rotating, two-finger pinch and zoom and three-finger flick. Currently, SGS is only supported under Windows but Synaptics is working on a Linux version.

Scrybe[20] is Synaptics latest product and enables gesture workflows. According to Synaptics, gesture workflows streamline multi-step tasks such as running presentations, doing research, watching DVDs and more—all by simply using the TouchPad.

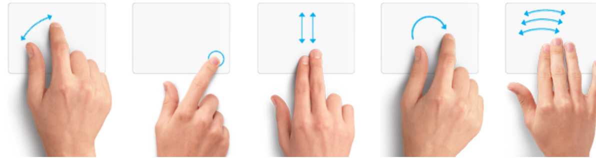


Figure 2.19: Gestures on the MacBook Multi-touch Trackpad

2.3 iGesture Basics

To completely understand the following chapters, more information about the architecture of iGesture and how iGesture works is needed. This section therefore introduces the basics of iGesture.

iGesture [26, 27] is a Java-based framework for gesture recognition and it is based on three main components and some common data structures as shown in Figure 2.20. These components are a recogniser, a management console and evaluation tools for testing and optimising algorithms.

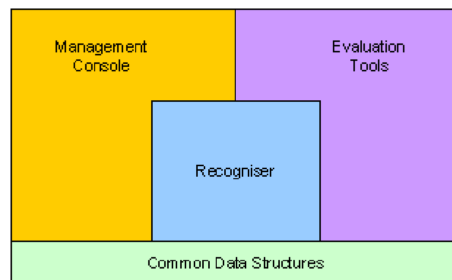


Figure 2.20: iGesture architecture

The common data structures define how the gestures are represented and grouped. Figure 2.21 shows the common data structures as a UML diagram. A `GestureClass` is an abstract representation of a gesture and gestures are grouped in a `GestureSet`. Since different algorithms need different descriptions of a gesture, the `GestureClass` itself does not contain any information describing the gesture. Instead, a descriptor is used to describe what the gesture looks like and any gesture description has to implement the `Descriptor` interface. Any `GestureClass` must have at least one descriptor and a name.

[19]http://www.synaptics.com/solutions/technology/gestures/touchpad
[20]http://www.uscrybe.com/index.html

The framework provides different descriptors including the `SampleDescriptor`, `TextDescriptor` and `DigitalDescriptor`. The `SampleDescriptor` describes the gesture by a number of samples and is therefore primarily used for training-based algorithms. A gesture sample is represented by the `GestureSample` class which contains the data captured by an input device. The `TextDescriptor` provides a textual description of the directions between characteristic points of a gesture. Whereas a `DigitalDescriptor` represents the gesture as a digital image. The digital descriptor is not used to recognise the gestures but rather to visualise a recognised gesture in a graphical user interface. The gesture classes and sets can be stored in XML format or in a db4objects[21] database, which is an open source object database for Java and .Net.

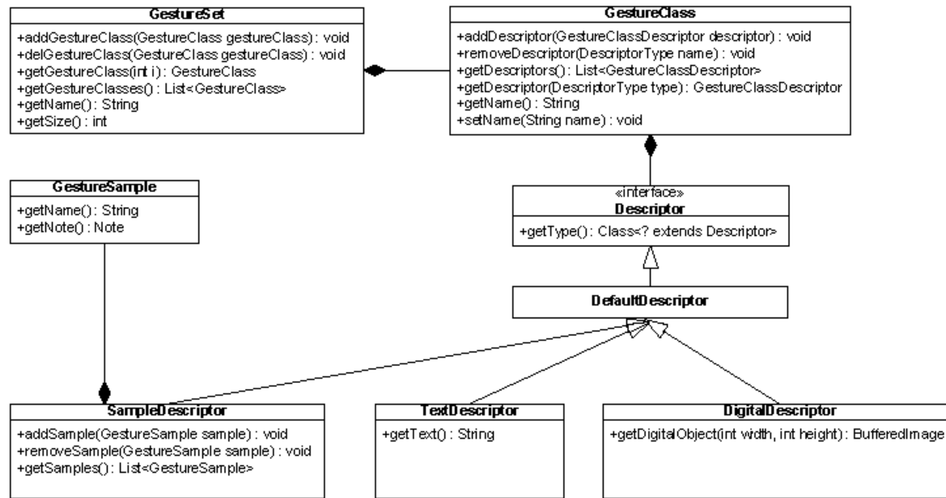


Figure 2.21: Gesture representation

The recogniser component can be configured to use different recognition algorithms. Each algorithm implements the `Algorithm` interface which is shown in Figure 2.22. This interface specifies methods for the initialisation, the recognition process, the registration of an event manager and for retrieving optional parameters and their default values.

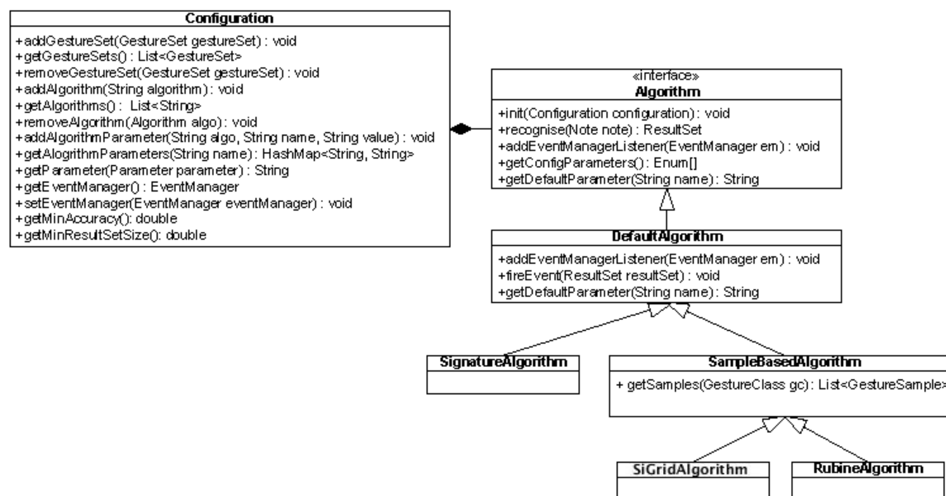


Figure 2.22: Algorithm class diagram

[21]http://www.db4o.com/

An algorithm always has to be initialised with an instance of the `Configuration` class containing gesture sets, an optional event manager and algorithm-specific parameters which are managed in a (key,value) collection. A new configuration can be created by using the Java API or an existing configuration can be loaded from an XML file.

iGesture currently offers four recognition algorithms: the Rubine and Extended Rubine algorithm, the Simple Gesture Recogniser Algorithm (SiGeR) and the SiGrid algorithm. The Rubine and SiGrid algorithms are sample-based recognition algorithms while the SiGeR algorithm is a signature-based algorithm that classifies gestures based on distance functions. Signer et al. describe these algorithms in detail in [27].

The `Recogniser` class, which is shown in Figure 2.23, can be configured with multiple algorithms. Depending on the method that is called, the recogniser will behave differently. The `recognise(Note note)` method runs through the algorithms in sequential order and stops the recognition process as soon as an algorithm returns a valid result. The `recognise(Note note, boolean all)` method executes all algorithms and combines all returned results. A `Note` is the data structure that contains the information captured by a 2D gesture input device. Each `Note` contains at least one stroke consisting of a list of timestamped positions. The `Recogniser` always returns a result set which contains an ordered list of result objects or empty if no gestures have been detected.

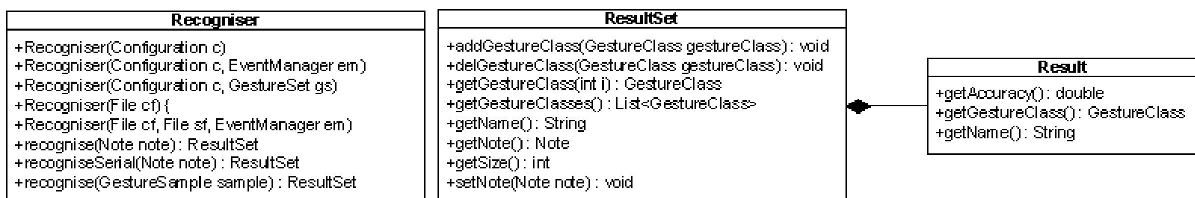


Figure 2.23: Recogniser API

iGesture provides support for different input devices like mice and tablets. In order to not depend on specific hardware, the `GestureDevice` interface was defined to enable a simple integration of new devices.

Applications that are interested in gesture events from gesture devices can register themselves as a `GestureEventListener` for a given device. If a gesture was performed, the device notifies all registered listeners. The listener then passes the captured gesture sample to a recogniser. After the recognition process, the recogniser notifies the event managers that registered themselves with the recogniser. An event manager implements the `GestureHandler` interface and specifies which action should take place when a certain gesture was recognised. Figure 2.25 shows a simplified view of the collaboration between the different components.

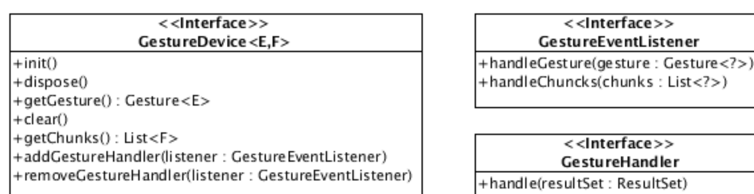


Figure 2.24: GestureDevice, GestureEventListener and GestureHandler interfaces

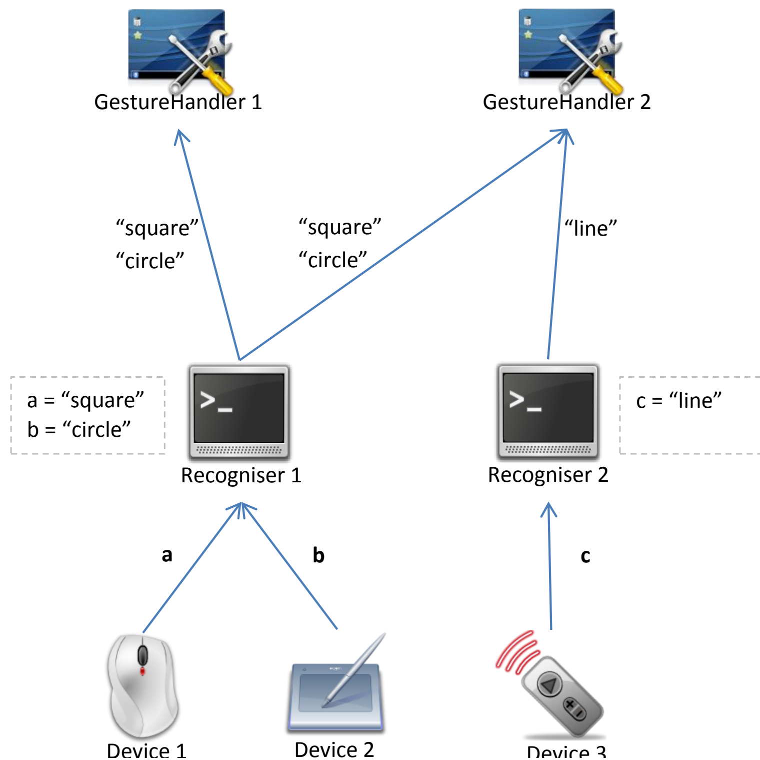


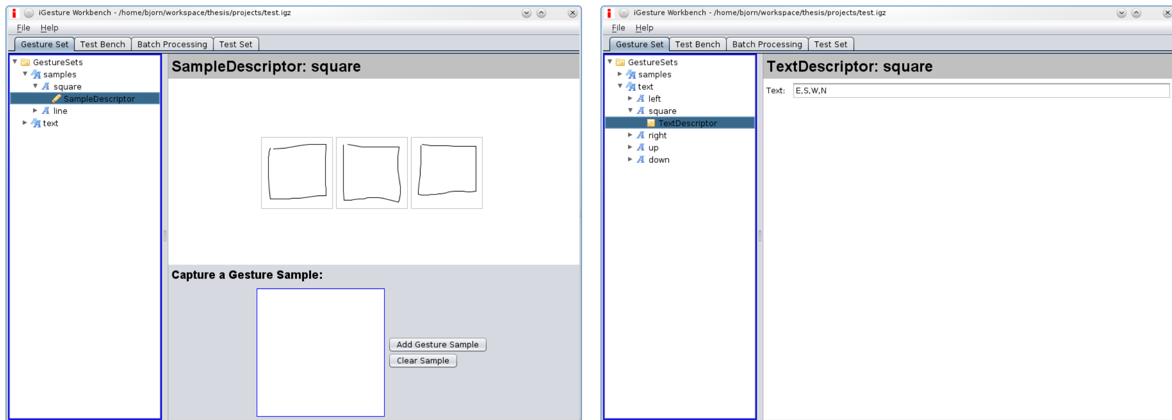
Figure 2.25: iGesture recognition architecture - overview

The iGesture framework also provides tools to define and test gestures. These tools are combined in the iGesture Workbench. The workbench contains four tabs: Gesture Set, Test Bench, Batch Processing and Test Set. The Gesture Set tab enables the user to manage gesture sets, gesture classes and descriptors. As mentioned earlier, there are different descriptors and therefore different representations of these descriptors. Figure 2.26a shows the sample descriptor panel. At the bottom of the panel, the input area is shown where a gesture sample captured from an input device is displayed. These samples can then be added to the sample descriptor and are shown in the top half of the panel. The representation of a text descriptor is shown in Figure 2.26b.

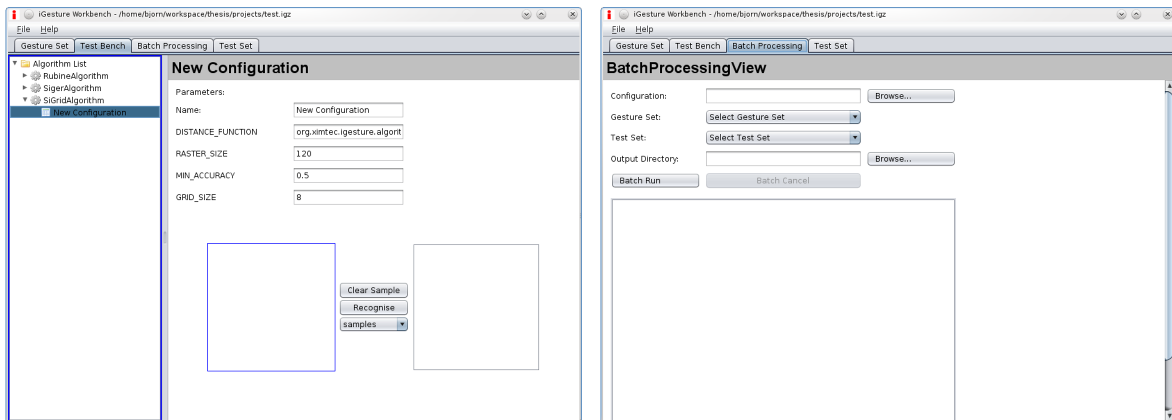
The Test Bench tab, which is shown in Figure 2.26c, provides the user with the functionality to capture a single gesture from an input device and to run the recogniser with a chosen gesture set and algorithm. This way, the recognition rate for a particular gesture set can be tested manually. In the Test Bench tab, a list of the available algorithms is shown. The user can create a configuration and configure the algorithm parameters shown in the upper half of the tab. The lower half shows the input area and the recognition results are displayed on the right-hand side of the input area.

The Test Set tab is used to manage and define test sets and gestures as shown in Figure 2.26e. These test sets are then used to test the algorithms in a batch process configured via the Batch Process tab (Figure 2.26d). To execute a batch process, the user first loads a configuration, then a gesture and a test set and finally specifies the location where the results of the batch process should be saved. The results itself are also displayed at the bottom of the tab.

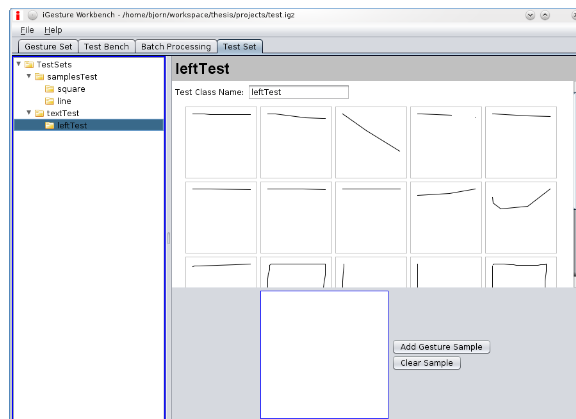
To summarise, iGesture provides support to design and test new recognition algorithms, tools to define new gestures and gesture sets and supports the integration of different input devices.



(a) Gesture Set tab with sample descriptor (b) Gesture Set tab with text descriptor



(c) Test Bench tab (d) Batch Processing tab



(e) Test Set tab

Figure 2.26: iGesture Workbench

2.3.1 Comparison with Multi-modal Fusion Engines

With the work presented in this thesis, multi-modal gesture interaction support is added to the iGesture framework. iGesture now supports multiple input modalities to define gestures including digital pen input, touch input (e.g. TUIO touch tables) and 3D gesture gestures (e.g. Wii Remote). Chapter 4 provides detailed information about the integration of the Wii Remote and TUIO devices in iGesture. However, iGesture does not provide multiple output modalities, whereas multi-modal fusion engines do support multiple input and output modalities.

Both approaches can be used to recognise composite gestures however they use different semantics. Multi-modal fusion engines try to identify what the user means at runtime while iGesture lets the user determine the semantics. We explain the difference by using the "Play next track" example that was mentioned in Section 2.1.3. In this example, the user gives the command "Play next track" and points to a track at the same time. In the case of multi-modal fusion engines, each atomic gesture has its own semantics. If the user gives the voice command, the next track in the list is played. If the user touches a track in the list, that track is played. When both commands are given at the same time, the meaning depends on how the fusion engine receives them. They can be interpreted as redundant and then only one track is played, while in the other case the next track in the list is first started and then the touched track is started.

In iGesture, it is up to the user to determine the semantics and the action to be performed. Independently of the order in which the multi-modal recogniser receives the recognised concurrent gestures (the "Play next track" voice command and the pointing gesture), the same action is performed. At the moment, we suppose that gestures are only composed in a single level. However, it is possible to support multi-level composites.

Chapter 3

A Multi-modal Gesture Recogniser

In Section 2.3 the basics of the iGesture framework have been introduced. We build upon some basic iGestures functionality to provide support for multi-modal and composite gesture interaction. First, we introduce the multi-modal recognition architecture and the different components. We then continue with the definition of the different kinds of multi-modal and composite gestures. This chapter is concluded with the description of the algorithm used by the multi-modal recogniser.

3.1 Multi-modal Recognition Architecture

We first summarise the normal recognition process shown in Figure 2.25. When a user performed a gesture with a particular device, the device sends the gesture sample to the registered event listeners. The `GestureEventListener` passes the sample to a `Recogniser` component and asks it to recognise the gesture. After the recognition process, the `Recogniser` sends the recognition results to the registered `GestureHandler`s. Based on the recognised gesture, the `GestureHandler` takes the appropriate action(s).

In the multi-modal recognition process, the output of the recognisers serves as input for the multi-modal recogniser. The multi-modal recogniser will continuously try to recognise multi-modal gestures while the `Recogniser`s work on a request-driven basis. Therefore, a component is needed to buffer the incoming gesture samples. Each multi-modal recogniser has its own buffer or queue where the gesture samples are ordered according to the start timestamp of the gesture sample.

Not all simple gestures are used to form composite gestures, so it does not make sense to send these gestures to the multi-modal recogniser. Those gestures will only fill up the queue and slow the whole recognition process down. To increase the efficiency of the multi-modal recogniser, an extra component is put between the `Recogniser`s and the multi-modal recogniser. This component, the multi-modal manager, only pushes the simple gestures that can be part of a composite into the queue while the other gestures are immediately sent to the registered listeners of the source `Recogniser` of that gesture sample. This way, the gestures that are not used to compose other gestures are delivered with a minimal delay.

Figure 3.1 shows a simplified view of the multi-modal recognition architecture. In this example, three devices are used to capture gestures. The devices send the captured gesture samples (a, b and c) to the recognisers. Each recogniser has been configured with a gesture set where each gesture is described using samples. Recogniser 1 recognises a “square” and “circle” while Recogniser 2 recognised a “line” gesture. Both recognisers work in multi-modal mode and send the recognition results to the multi-modal manager. The manager is linked to a multi-modal recogniser which has been configured

with a gesture set with a single composite gesture: "concurrent". The manager knows all the different composing gestures and can now filter its input. The "square" and "circle" gestures potentially form part of a composite gestures and the manager forwards them to the input queue of the multi-modal recogniser. The "line" gesture however is not part of any composite gesture and the recognition result is sent to the gesture handlers that were registered with the originating recogniser Recogniser 2. In this example, only GestureHandler 2 is notified about the "line" gesture.

The multi-modal recogniser recognises the "concurrent" gesture since the "square" and "circle" gesture were performed in parallel. Then the multi-modal recogniser notifies all registered handlers. In this case, both GestureHandler 1 and 2 registered themselves with the multi-modal recogniser.

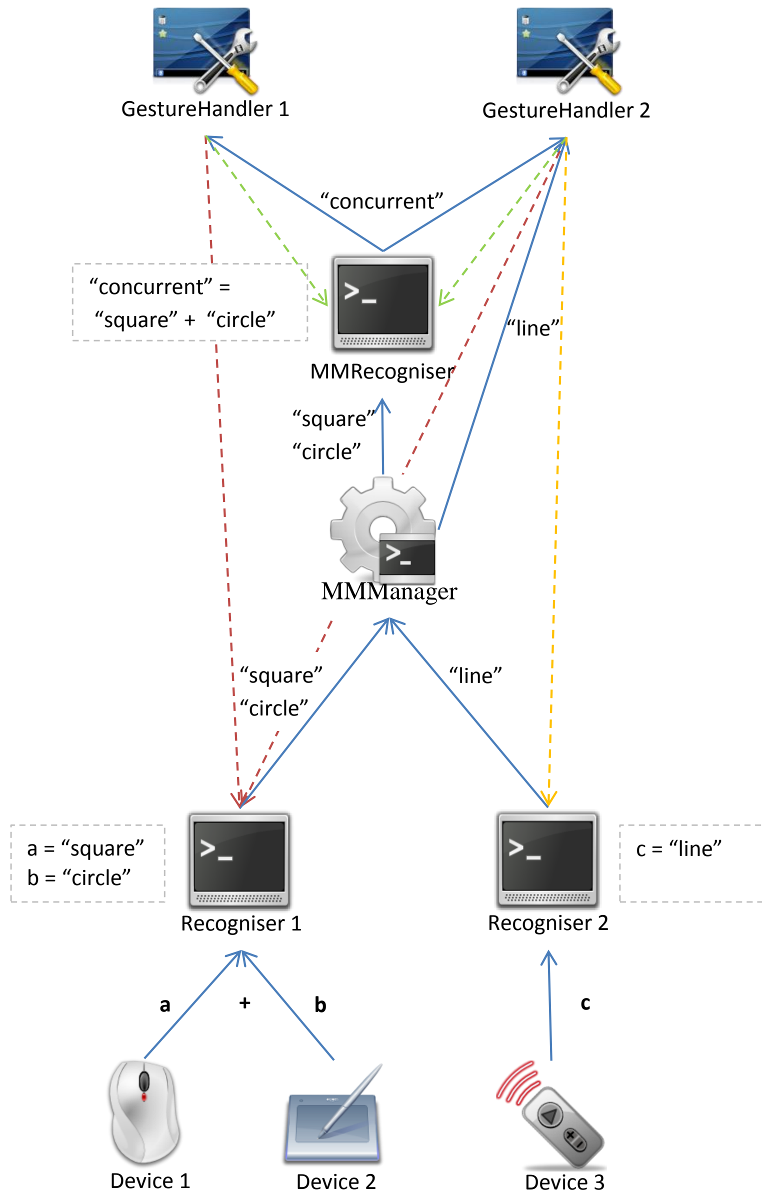


Figure 3.1: Multi-modal recognition architecture - multi-modal mode

To summarise, the behaviour of a `Recogniser` varies depending on its mode. In normal mode, the `Recogniser` is not associated with a multi-modal manager and sends the recognition results immediately to the registered `GestureHandlers`. In multi-modal mode, the `Recogniser` is associated with a multi-modal manager and sends the recognition results only to that manager.

There is also a third mode which combines both behaviours, hence the name mixed mode. In mixed mode, the `Recogniser` sends the recognition results to the multi-modal manager and to the registered `GestureHandler`s. In Figure 3.2 both `Recogniser`s are in mixed mode. This means that, for example, Recogniser 2 sends the "line" gesture to the multi-modal manager and to GestureHandler 2. A side-effect of the mixed behaviour is that the `GestureHandler` may receive the same gesture twice. Therefore, it is up to the multi-modal manager to make sure this does not happen. If the gesture is not used to compose other gestures (e.g. the "line" gesture), the multi-modal manager checks if the source recogniser is in mixed mode or not. Since Recogniser 2 is in mixed mode, the manager does not send the "line" gesture to GestureHandler 2 since it already received it. If Recogniser 2 is in multi-modal mode, the manager does send the gesture to GestureHandler 2.

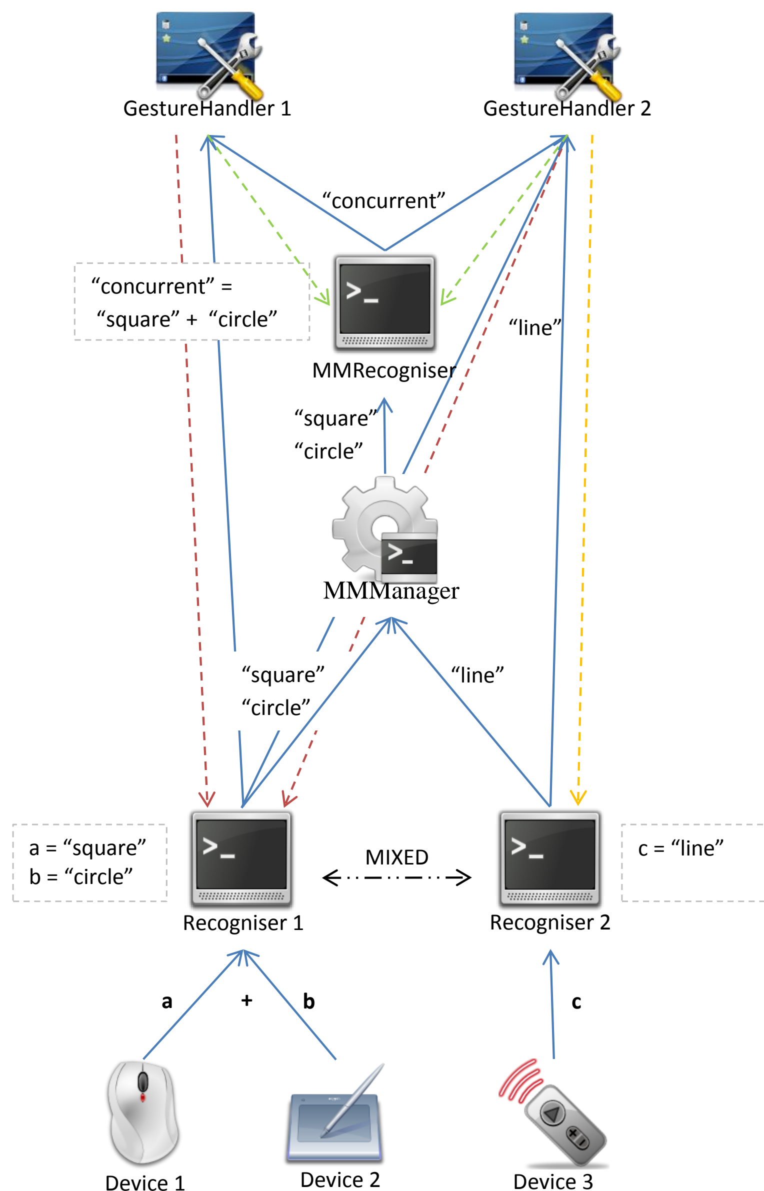


Figure 3.2: Multi-modal recognition architecture - mixed mode

Another problem arises when a `GestureHandler` listens to both the composite gestures and the gestures that form that composite. This is for example the case if GestureHandler 1 would listen to the "square" gesture and to the "concurrent" gesture. In this case, GestureHandler 1 first receives

the "square" gesture and then the "concurrent" gesture which actually uses the same instance of the "square" gesture which is not the desired behaviour. While in multi-modal mode, the single "square" gesture will be therefore delayed by the multi-modal recogniser—supposing that no "circle" gesture was made at the same time— and only returned as a simple gesture if the composite gesture has not been detected.

If one `GestureHandler` is not registered with the multi-modal recogniser and another `GestureHandler` is, another side-effect may occur. In multi-modal mode, the non-registered `GestureHandler` will not receive the gesture that potentially forms part of a composite gesture, while the registered `GestureHandler` does. This could be an undesired behaviour.

The mixed behaviour makes it possible to send recognised simple gestures to the multi-modal recogniser and at the same time to a `GestureHandler` that is not registered with multi-modal recogniser since it is not interested in composite and multi-modal gestures. However, the configuration of the `GestureHandler`s is crucial. Thus, in general it is not advised to use gestures that might form part of a composite (e.g. "square" and "circle") on their own.

3.2 Composite Gesture Definition

A multi-modal or a composite gesture is described via a composite descriptor. A composite descriptor contains a constraint and each constraint can be composed of an unbounded number of gestures. We have defined six main types of composite gesture constraints:

- concurrency constraint
- sequence constraint
- proximity and concurrency constraint
- proximity and sequence constraint
- interval constraint
- cardinality device constraint
- cardinality user constraint

A concurrency constraint defines a composite of two or more simple gestures that are performed in parallel. All gestures that form part of the composite must have a common overlap in time in order to fit the constraint. Two or more gestures that are executed in sequential order constitute a sequence constraint. The minimum and maximum time between every two consecutive gestures must be defined.

The proximityconcurrency and proximitysequence constraint are variations on the two previous constraints. These proximity-based constraints add a distance parameter. For 2D gestures, this distance is the length of the diagonal of the combined bounding box of the gestures that form part of the composite gesture (x in Figure 3.3). In the case of 3D gestures, the distance parameter defines the diagonal of the largest face of a 3D rectangular bounding box (y in Figure 3.3). The minimum and the maximum value for the distance parameter have to be defined. The distance unit has to be specified as well (e.g. m, cm, mm or km).

A variation on the sequence and proximitysequence constraint allows to specify different values for each gap between two consecutive gestures. For example, minimum 5 and maximum 30 seconds between gesture 1 and 2 while between gesture 2 and 3 there is a gap of minimum 3 seconds and maximum 15 seconds.

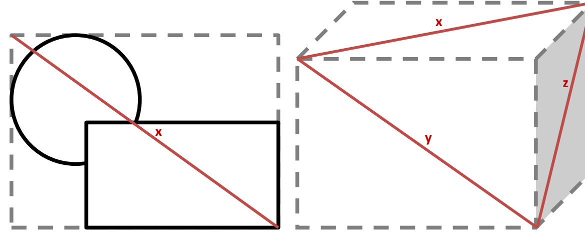


Figure 3.3: Distance parameter of the proximity constraint

In some cases, it might be difficult to predict the temporal behaviour of multiple gestures. It is possible that they are performed in parallel or in sequence or in combination of the two. Another constraint, the interval constraint, was defined to cope with this kind of behaviour. Within a certain time interval, the specified gestures must be performed but the order does not matter.

A variation on the interval constraint are the cardinality based constraints. As the name suggests, the associated action will only be performed if a specified threshold is reached (e.g. 2 out of 7). Within a certain time interval, a particular gesture must be performed at least a minimum number of times. The maximum must also be specified. In a cardinality device constraint each device gets one vote, while in a cardinality user constraint each user gets one vote. A cardinality constraint can be used to specify any kind of majority (e.g. $50\% + 1, \frac{2}{3}$).

It is also possible to specify for each gesture that forms part of a composite, by which user and by what kind of device that gesture should be performed. If a device type is specified, specific devices can be specified as well by their unique identifiers. Note that for a cardinality constraint the user is not specified.

3.3 Multi-modal Recognition Algorithm

The algorithm used by the multi-modal recogniser is based on string pattern-matching. The algorithm is illustrated as a flowchart in Figure 3.4. First, a unique character is associated with each kind of gesture that forms part of a composite gesture, for example "a" for "square" and "b" for "circle". Based on these character representations, the composite gesture is represented by one or more patterns. A sequence constraint has a fixed order and as a consequence only one pattern is needed to represent the constraint. A composite gesture where a "square" is followed by a "circle", is represented by the pattern "ab".

However, for a concurrency constraint, it is impossible to know the order in which the gestures are going to be present in the queue. Concurrent gestures never start at exactly the same time, and even if this would be the case, the order in the queue depends on the time when a gesture is added to the queue. So for concurrent gestures, all permutations must be generated. A composite gesture where a "square" and a "circle" are performed in parallel, is therefore represented by the patterns

"ab" and "ba". This is also the case for interval and cardinality constraints since the order of the composing gestures is unspecified. For cardinality constraints this does mean that the number of patterns increases exponentially. The number of patterns would increase according to the following formula:

$$\sum_{i=k}^{n} = k!\binom{n}{k} \begin{cases} \text{where } n \text{ is the maximum occurence} \\ \text{where } k \text{ is the minimum occurence} \end{cases}$$

In general, from a group of n characters, $\binom{n}{k}$ different groups of k characters can be chosen and each pattern of k characters has k! variations.

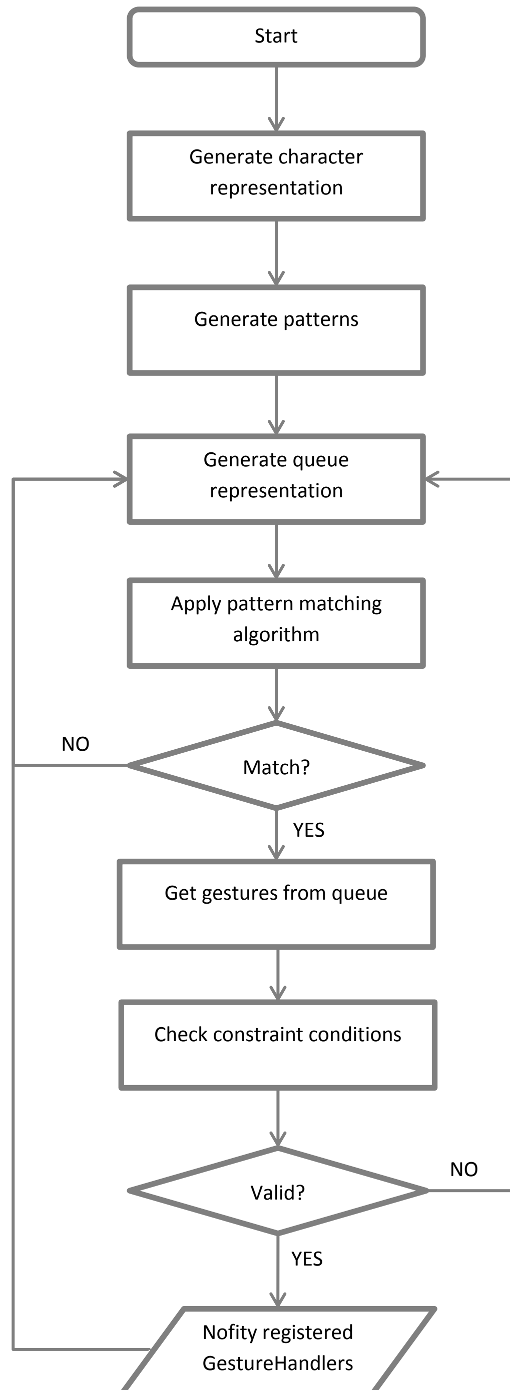


Figure 3.4: Flowchart of the multi-modal recognition algorithm

For example, for a cardinality constraint composed out of 5 different gestures of which minimum three gestures must be performed, 300 patterns are required.

$$\begin{aligned}\sum_{i=3}^{5} &= 3!\binom{5}{3} + 4!\binom{5}{4} + 5!\binom{5}{5} \\ &= \frac{5!}{3!(5-3)!} + \frac{5!}{4!(5-4)!} + 5! \\ &= 3!10 + 4!5 + 5! \\ &= 60 + 120 + 120 = 300\end{aligned}$$

Therefore, we decided that a cardinality constraint should be defined only with a single type of gesture. For the example mentioned above, we require only three patterns: "ccc", "cccc" and "ccccc".

Once all patterns are known, the recognition process can begin. A string representation is created from the gesture queue and then a pattern matching algorithm is applied. Knuth-Morris-Pratt (KMP) [18] and Boyer-Moore (BM) [9] are the best known string search algorithms. However, these algorithms can only find exact matches. Since multiple gestures can be performed at the same time, a simple gesture can end up in the queue between two gestures that form a composite. In that case, the pattern will not be found. For example, a sequential composite gesture—"square" followed by "circle"— is performed by one user and at the same time another user performs the gesture "line". Because the gestures are ordered on a temporal basis in the queue, the resulting queue is:

"square" - "line" - "circle"

To deal with multiple parallel gestures we need a fuzzy string search algorithm. The Bitap [1] algorithm by Baeza-Yates and Gonnet will be used. The algorithm is also known under the name Shift-And or Shift-Or. This algorithm can search for a pattern allowing for certain errors, where errors can be inserted, removed or replaced characters. The algorithm uses bitwise techniques and is very efficient for relatively short patterns. For a pattern length m, a text length n and alphabet size σ, the time complexity is $O(m + \sigma)$ for the preprocessing phase and $O(n)$ for the searching phase. The Levenshtein[1] distance is used to define the number of allowed errors. The algorithm is often used to find the differences between two blocks of text.

If a potential match is found, the corresponding gestures are retrieved from the queue and the constraint conditions are checked. If the conditions are valid, the composite gesture was found and the registered `GestureHandler`s are notified.

There are still two issues that have to be discussed. Firstly, it is unnecessary to always create a string representation of the complete queue and secondly, items must be removed from the queue as well. A sliding time window approach is used to solve both problems. For each of the composing gestures, the maximum time window is determined. If, for example, the "square" gesture is used in a concurrency constraint and in a sequence constraint with a gap of maximum 30 seconds, then the maximal time window for "square" is about 35 seconds.

$$tw_{square} = max(t_{gesture} + t_{processing},\ t_{gesture} + t_{processing} + t_{gap}) = 3 + 0.2 + 30 = 33.2\ seconds$$

[1]The Levenshtein distance between two strings is defined as the minimum number of edits that are needed to transform one string into the other, with the allowable edit operations being insertion, deletion or substitution of a single character. http://en.wikipedia.org/wiki/Levenshtein_distance

This time window consists of the time to perform the gesture, the estimated processing time of the recognition process and the time between two consecutive gestures, which is a parameter of the sequence constraint. All gestures within a window of 35 seconds before and after the "square" gesture are used to create a string representation.

Gestures in the beginning of the queue that are not covered by a time window and have not been recognised as part of a composite gesture are removed from the queue and the `GestureHandler`s that are registered with the source `Recogniser` are notified. Gestures that have been recognised as part of a composite gesture and that are not covered any more by a time window are removed as well.

Chapter 4

Implementation

In Chapter 3, the multi-modal recogniser concepts have been introduced. This chapter primarily deals with the implementation of the multi-modal recogniser. Another subject that is discussed is the device manager and how it supports the flexible addition, removal and coupling of users and devices at runtime. Furthermore, we address the support of additional gesture devices as well as the persistent storage of composite gestures.

4.1 Device Manager

Multi-modal gestures are performed with multiple devices by one or more users. To manage all these different devices and users, a specific device manager component that is shown in Figure 4.1 has been developed.

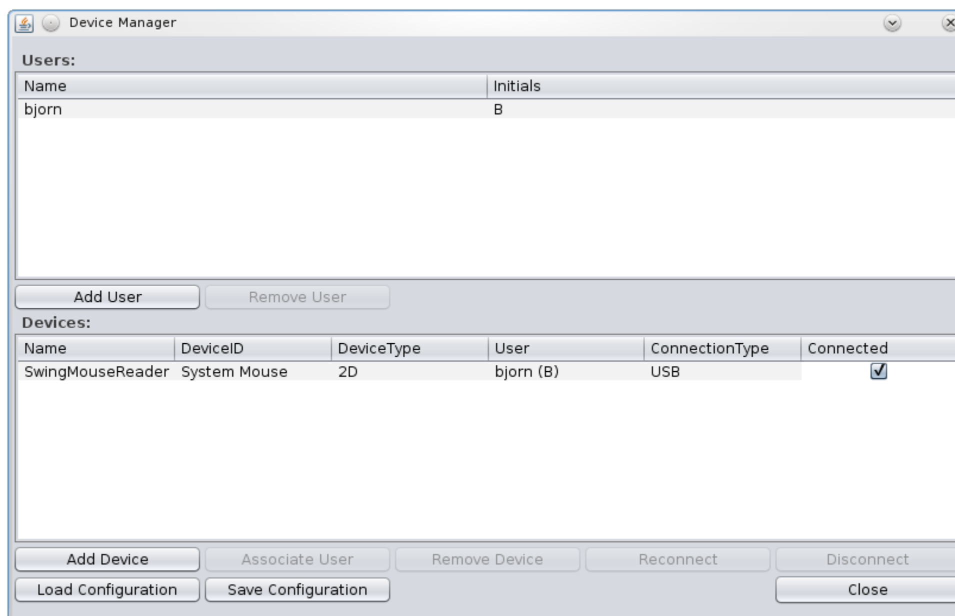


Figure 4.1: Device Manager

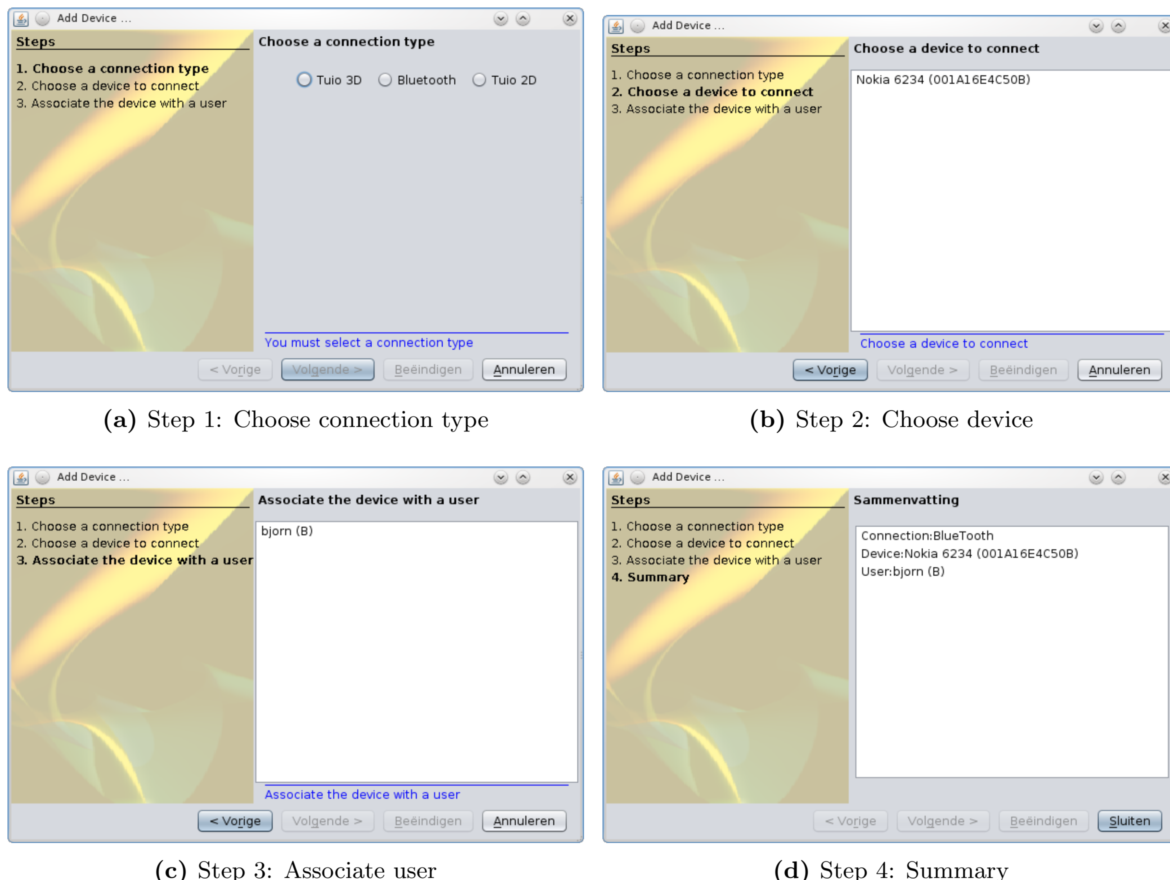
4.1.1 Functionality

The device manager offers several functions. The top half of the window provides an overview over the users registered with the application. For each user, the name and initials are displayed. The initials are used to disambiguate users with the same name. By default, a system user is added to the list and this default user cannot be removed. New users can be added and registered users can be removed as well. To add a new user, the name and the initials of the user are entered in a simple dialogue box.

In the bottom half of the window, the active devices together with their associated users are displayed. For each device, the name, a unique identifier, the type of the gestures to be performed by the device (e.g. 2D, 3D, voice), the associated user, the connection type and the connection status are shown. By default, the system mouse is added to the list and cannot be removed.

It is possible to add new devices as well as remove active devices. Furthermore, a different user can be associated with a device via a dialogue that shows a list of active users. If a user is removed from the device manager, the devices associated with that user are automatically associated with the the system user.

A wizard with multiple steps is used to add new devices. In the first step, the user chooses a connection type as shown in Figure 4.2a. The corresponding discovery service then tries to detect available devices. Any detected device is added to the list in the second step as highlighted in Figure 4.2b. The user chooses the desired device and can associate the device with a user of their choice in the next step. A summary is shown in the last step as illustrated in Figure 4.2d.



(a) Step 1: Choose connection type

(b) Step 2: Choose device

(c) Step 3: Associate user

(d) Step 4: Summary

Figure 4.2: Add Device wizard

If users have a specific setup for their devices, it can be a tedious task to manually add these devices one by one each time at startup. Therefore, the device manager also provides the possibility to save and load the (device,user) configurations. When such a configuration is loaded, the device manager automatically tries to reconnect to all the devices. If it fails to connect, for example because a Bluetooth device is not discoverable, the user can still manually reconnect after making the device discoverable. Note that the device manager is available as a service within the iGesture Workbench to enable universal access to the device manager.

4.1.2 Implementation

To allow developers to create their own implementation of the device manager, the following five interfaces have been defined:

- an interface for the users (`IUser`)
- a device interface (`Device`)
- a device discovery service interface (`DeviceDiscoveryService`)
- two device manager interfaces (`IDeviceManager` and `IDeviceManagerView`)

IUser Interface

The `IUser` interface shown in Listing 4.1 defines all methods a user class should provide. Users have a name and initials for unique identification. To ensure the uniqueness of the initials, the device manager checks whether the initials are already in use or not. If the initials are used, the user is asked to enter alternative initials. The system user is the default user which cannot be removed from the device manager.

Listing 4.1: IUser Interface

```
public interface IUser {

   String getName();
   void setName(String name);

   String getInitials();
   void setInitials(String initials);

   String toString();
}
```

Device Interface

Devices should be uniquely identifiable based on an identifier such as a MAC address. A device also needs a human readable name since the unique identifiers are normally difficult to remember. Furthermore, each device has a gesture type and connection type. The gesture type can for example be voice or refer to the dimension of the gestures that are performed with the device(e.g. 2D or 3D). An example of a connection type is Bluetooth. The device class indicates what kind of device it is and corresponds to the class name, for example `WiiReader` for a Wii Remote.

Functionality to connect to and disconnect from a device is necessary too. A final important characteristic of a device specifies whether it is a mandatory device or not. The system mouse is an example of a mandatory device. Mandatory devices cannot be removed from the device manager. The `Device` interface is shown in Listing 4.2.

Listing 4.2: Device interface

```
public interface Device {

    // device identifier
    String getDeviceID();
    void setDeviceID(String id);

    // display name
    String getName();
    void setName(String name);

    //allow management of the connection with the device
    void disconnect();
    void connect();
    boolean isConnected();
    void setIsConnected(boolean isConnected);

    //type of the device, e.g. 2D, 3D, voice
    int getDeviceType();
    void setDeviceType(int deviceType);

    //type of the connection, e.g. USB, BlueTooth
    int getConnectionType();
    void setConnectionType(int connectionType);

    String getDeviceClass();

    boolean isMandatoryDevice();
    void setMandatoryDevice(boolean isMandatory);

    String toString();
}
```

DeviceDiscoveryService Interface

While adding a device, device discovery services are used to search for devices. A mapping between the connection types and the corresponding discovery services is defined in an XML file. In this way new connection types can be added in a flexible way without major changes to the framework. An example of such a configuration file is shown in Listing 4.3.

Listing 4.3: connections.xml

```
<connections>
 <connection>
  <name>Bluetooth</name>
  <discoveryService>
   org.ximtec.igesture.discoveryservice.BluetoothDeviceDiscoveryService
  </discoveryService>
 </connection>
 <connection>
  <name>Tuio 2D</name>
  <discoveryService>
   org.ximtec.igesture.discoveryservice.Tuio2DDeviceDiscoveryService
  </discoveryService>
 </connection>
</connections>
```

For each connection, the name of the connection type and the class of the corresponding discovery service is mentioned. When the device manager is initialised, the configuration file is read and a mapping is created.

Each discovery service provides a method to discover devices and a method to cleanup the service. The `discover()` method instantiates a subclass of `AbstractGestureDevice<E,F>` for each discovered device as shown in Listing 4.4. To know what specific class should be instantiated, an extra configuration file is used which specifies which class corresponds to the device class. Examples of device configurations are discussed in 4.2.1 and 4.2.2.

Listing 4.4: DeviceDiscoveryService interface Interface

```
public interface DeviceDiscoveryService {

   //Discover and return the found devices.
   public Set<AbstractGestureDevice<?,?>> discover();

   //Clean-up the discovery service.
   public void dispose();
}
```

IDeviceManager Interface

The device manager consists of the controller and view components. The controller component is defined by the `IDeviceManager` interface. The most important features of the device manager are adding and removing users, adding and removing devices and associating a user with a device. A configuration of users and devices can be saved to and loaded from a file. This enables the reuse of a configuration without having to manually add each device and user. As mentioned earlier, a mapping between the connection types and the corresponding device discovery services is kept in the device manager. The `getDiscoveryMapping()` method can be used to retrieve this mapping. It should further be possible to get the default user as well as all the users and devices registered with the device manager as shown in Listing 4.5.

Listing 4.5: DeviceManager interface

```
public interface IDeviceManager {

    void addUser(User user);
    void removeUser(User user);

    void addDevice(AbstractGestureDevice<?,?> device, User user);
    void removeDevice(AbstractGestureDevice<?,?> device);

    void associateUser(AbstractGestureDevice<?,?> device, User user);
    IUser getAssociatedUser(AbstractGestureDevice<?,?> device);

    User getDefaultUser();

    Set<AbstractGestureDevice<?,?>> getDevices();
    Set<User> getUsers();

    Map<String, DeviceDiscoveryService> getDiscoveryMapping();

    void saveConfiguration(File file);
    void loadConfiguration(File file);
}
```

Listing 4.6: DeviceManagerView interface

```
public interface IDeviceManagerView {

    public void addUser(User user);
    public void removeUser();

    public void addDevice(DeviceUserAssociation association);
    public void removeDevice();

    public Collection<DeviceUserAssociation> getDevices();

    public void updateDevice(Object value, int column, DeviceUserAssociation o);

    public DeviceUserAssociation getSelectedDevice();
    public User getSelectedUser();

    public void clear();
}
```

IDeviceManagerView Interface

The view of the device manager is defined by the `IDeviceManagerView` interface shown in Listing 4.6. The view offers similar functionality as the controller such as the addition and removal of users or devices. It is further possible to update the view and get the selected user or device. A `DeviceUserAssociation` defines a relationship between a device and a user. For further information about the device manager, the reader should consult Appendix A.1 which contains UML diagrams of all the mentioned classes.

4.2 Integration of New Gesture Devices

In order to demonstrate and verify the operation of the multi-modal recogniser, the user should be able to perform specific gestures. Therefore, support for gesture devices has to be provided. iGesture already supports mice and digital pens. To offer the user alternative choices, we decided to add the Wii Remote as well as TUIO devices as extra devices.

4.2.1 Integration of the Wii Remote in iGesture

As part of a student project, Arthur Vogels [31] investigated the support for 3D gesture recognition in iGesture based on the Wii Remote. Vogels made a separate tool to demonstrate the feasibility of 3D gesture recognition. Our integration of the Wii Remote is based on his research and conclusions.

We first introduce the necessary software and then discuss the implementation and integration specific details. Finally, we provide some initial results for the performance of the proposed 3D gesture recognition.

Software

The Wii Remote, often referred to as wiimote, uses Bluetooth to connect with the Wii console. To communicate with the Wii Remote from a PC, a JSR82[1] compatible library is needed. JSR82 is a specification that describes a set of Java Bluetooth APIs defined by the Java Community Process Program. Vogels mentioned that BlueCove[2] and Avetana[3] are the two most widely used implementations.

BlueCove is available for free via an Apache 2.0 license. Avetana is freely available only for Linux under a GPL 2.0 license, whereas a fee has to be paid for Windows and Mac OS X. Since both implementations support all three main operating systems, the former implementation was chosen based on the fact that it is freely available for all operating systems.

Note that Linux users have to use an extra module due to some licence issues. This module is available under a GPL license and makes it possible to use the BlueZ[4] Bluetooth stack, which is the default Linux Bluetooth stack. Note that the used Bluetooth stack must support the L2CAP protocol in order to be able to communicate with a Wii Remote.

While the BlueCove library enables the set up of a Bluetooth connection with an arbitrary Bluetooth device, a library to communicate with the Wii Remote and interpret the data from the Wii Remote is also necessary. In his project, Vogels researched the WiiGee[5] and Motej[6] libraries which

[1] http://jcp.org/en/jsr/detail?id=82
[2] http://bluecove.org/
[3] http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml
[4] http://www.bluez.org/
[5] http://www.wiigee.com/index.html
[6] http://motej.sourceforge.net/index.html

are both implemented in Java. There exist a number of other libraries that support the Wii Remote, each with its own features and characteristics. Therefore, more detailed comparison was required since some of the libraries might have changed over time. The other discussed libraries are:

- libwiimote (http://libwiimote.sourceforge.net/)
- JWiiPIE (http://sourceforge.net/projects/jwiipie/)
- WiiMoteCommander (http://wiimotecommande.sourceforge.net/)
- WiiUseJ (http://code.google.com/p/wiiusej/)
- openWiiMote (http://code.google.com/p/openwiimote/)
- CWiid (http://abstrakraft.org/cwiid/)
- WiiMote-Simple (http://code.google.com/p/wiimote-simple/)
- WiiRemoteJ (http://www.world-of-cha0s.hostrocket.com/WiiRemoteJ/)

On the wiili[7] and wiibrew[8] websites, some other platform specific libraries are mentioned. Most of the discussed libraries support the use of the buttons, the accelerometer, the LEDs, the infrared functionality and the vibration feedback (also referred to as rumble). Most of the libraries can also read and write to the EEPROM and registers of the WiiMote. The only exception is openWiimote which does not support the accelerometer, the infrared and rumble capabilities. Some of the libraries support reading the battery status and using the sound feedback on the controller. Furthermore, not all libraries are available for all three major operating systems (Windows, Linux and Mac OS X) and some of them do not support the extensions of the Wii Remote. Table 4.1 shows a comparison of the different libraries (last checked in May 2010).

Our decision for a specific library was based on a number of factors. First of all, the library should be platform independent. Since the iGesture framework can be used on Windows, Linux and on Mac OS X, the support for the Wii Remote should be available as well on all these platforms. Furthermore, to keep the framework simple and clear, only one library should be used to provide support for the Wii Remote on all three major platforms. Secondly, the library should be open source, so that the source code can be adapted if necessary. Recent development activity and therefore an active project was the third important criteria. In addition, support for the accelerometer is critical, since the data obtained from the accelerometer is used to train and recognise the gestures. The possibility to use the Wii MotionPlus extension was further seen as a plus. Based on the criteria mentioned above, we decided to use the WiiGee library.

[7] http://www.wiili.com/index.php/Main_Page/
[8] http://wiibrew.org/wiki/Main_Page/

Table 4.1: Comparison of different Wii Remote libraries (May 2010).

Name	Version	Motion Plus	Nunchuck	Linux	Mac OSX	Windows	Language	Multiple WiiMotes	License	Needs BlueCove
WiiGee	1.5.6	Read	No	Yes	Yes	Yes	Java	Yes	LGPL v3	Yes
Motej	0.9	No	Yes	Yes	Yes	Yes	Java	N.A.	Apache 2.0	Yes
libwiimote	0.4	No	Yes	Yes	No	No	C	N.A.	GPL v2.0	No
JWiiPIE	0.0.2	No	No	Yes	Yes	Yes	Java	Yes		Yes
WiiMoteCommander	2.0	No	Yes	Yes	Yes	Yes	Java	Yes	LGPL v2.1	Yes[a]
WiiUseJ	0.12b	No	Yes	Yes	No	Yes	Java	Yes	LGPL v3	No[b]
openWiiMote	1.0	No	No	Yes	Yes	Yes	Java	N.A.	LGPL v3	Yes[c]
CWiid	0.6	In progress	No	Yes	No	No	C	N.A.	GPL v2.0	No
WiiMote-Simple	1.0.1	No	No	Yes	Yes	Yes	Java	N.A.	LGPL v3	Yes
WiiRemoteJ	1.6	No	Yes	Yes	Yes	Yes	Java	Yes		Yes

[a] needs WiiRemoteJ API and Java 3D
[b] Java wrapper for the wiiuse library (written in C)
[c] depends on log4j

Implementation

The Wii Remote is connected through a Bluetooth connection. This means that it is discovered by the Bluetooth Device Discovery Service. This service instantiates a subclass of `AbstractGestureDevice` for each discovered device. Of course, the service needs to know which device corresponds to which class. Therefore a mapping is defined in an XML configuration as shown in Listing 4.7.

Listing 4.7: bluetoothdevices.xml

```
<devices>
  <device>
    <minor>4</minor>
    <major>1280</major>
    <name>Nintendo RVL-CNT-01</name>
    <class>org.ximtec.igesture.io.wiimote.WiiReader</class>
  </device>
</devices>
```

Each Bluetooth device has a major and a minor device class as well as a name. The major device class specifies the main classification of the device (e.g a computer (256), a peripheral (1280) or a phone (512)). The minor device class is a more specific description of the device and has to be interpreted in the context of the major class. For example, in the case where the major class is a computer, the minor device class can have the values of a laptop or a desktop.

However, it is possible that a device is not classified. This means that both the minor and the major device class are set to 0. In that case, the device name is used to identify the device. This implies that we need a separate entry in the XML configuration for each name related to different devices of a given type.

The `class` node denotes the subclass that will be instantiated using Java Reflection. The `WiiReader`, `WiiReaderPanel` and other classes to support 3D gestures in general and more specifically the Wii Remote were taken over from Arthur Vogels and adapted where necessary. For example, the `WiiReader` class now implements the `Device` interface to make it compatible with the device manager.

It is possible to graphically represent a gesture in a GUI using a panel. To support the automatic visualisation of a gesture in the panel, the panel must implement the `GestureEventListener` interface shown in Listing 4.8. When a gesture has been performed, the device notifies all registered listeners and the panel can render the gesture.

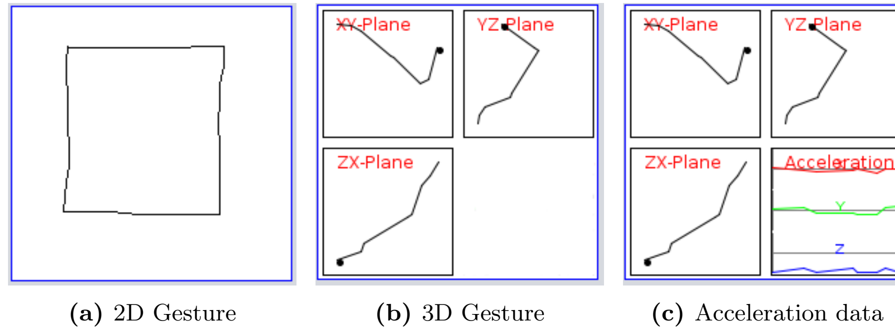
Listing 4.8: GestureEventListener Interface

```
public interface GestureEventListener {

  void handleGesture(Gesture<?> gesture);

  void handleChunks(List<?> chunks);
}
```

Of course, the representation of a two-dimensional gesture is different from the representation of a three-dimensional gesture. This means that the `paintComponent()` method of the panel differs for each representation format. Figure 4.3a shows an example for a two-dimensional gesture and 4.3b and 4.3c for a three-dimensional gesture without or with acceleration data.

Even each device can have its own representation format for the gestures that can be performed with the device and therefore its own rendering panel. To get the panel that corresponds to a specific

device, the `getPanel()` method can be used. All these panels have `GestureDevicePanel` as a common superclass. `GestureDevicePanel` is an abstract class that extends `JPanel` and implements the `GestureEventListener` interface.



(a) 2D Gesture **(b)** 3D Gesture **(c)** Acceleration data

Figure 4.3: GestureDevicePanel Examples

The drawing methods from the `WiiReaderPanel` class where moved to the `Note3DTool` class to provide a default representation for `Note3Ds`. The visualisation of a 2D gesture is shown in Figure 4.3a, while a 3D gesture is highlighted in Figure 4.3b and 4.3c.

Recognition Algorithm

To be able to recognise a 3D gesture, a 3D gesture recognition algorithm is needed. So far, iGesture only supports 2D gesture recognition algorithms. Vogels [31] presented an algorithm based on the Extended Rubine algorithm which is already used by iGesture to recognise 2D gestures [23, 27]. The data captured from the 3D input device is split into the XY, YZ and XZ planes and the Extended Rubine algorithm is applied to the data in each of these planes.

We use a trigger to clearly mark the beginning and the end of a gesture. In this way, the recogniser receives only the relevant acceleration data since the Wii Remote transmits acceleration data at the slightest movement.

Table 4.2 shows the results of some initial tests. We tested a set of six gestures in a random order with the ERubine algorithm. Each test person performed the gestures ten times. Only the first five of the twenty-one features were used and the recognition rates are already quite promising. However, more elaborate testing will have to be done.

The “triangle” gesture was recognised every time and the “Z”, “in/out” and “up/down” gestures have acceptable recognition rates although the “Z” gesture did not produce any recognition result in one case. The two remaining gestures do not have an acceptable recognition rate. Holding the Wii Remote in a slightly different way influences the acceleration data significantly. As a consequence, the captured data differs strongly from the data of the training samples leading to unacceptable recognition rates. Using the Wii MotionPlus extension could help to improve the results by taking the orientation of the Wii Remote into account.

In literature, a variety of other projects that use the Wii Remote or other 3D gesture devices can be found. These projects use more sophisticated recognition algorithms that are based on Support Vector Machines [20], Neural Networks [20], Hidden Markov Models [19, 24, 34], Dynamic Time Warping [20, 34] or Linear Time Warping [34]. Implementations of many of these algorithms can be found in the WEKA[9] tool [12], which is a Java-based open source machine learning workbench.

[9]http://www.cs.waikato.ac.nz/ml/weka/

Table 4.2: Rubine 3D recognition results

Gesture	Correct	False	Not	Recognition rate
triangle	20	0	0	100%
diagonal	13	7	0	65%
circle	11	9	0	55%
Z	17	2	1	85%
in/out	18	2	0	90%
up/down	18	2	0	90%

4.2.2 Integration of TUIO in iGesture

Besides the integration of the Wii Remote, support for TUIO devices has been added to iGesture as well. We first briefly introduce the TUIO protocol and then describe the implementation and integration of a TUIO client in the iGesture framework. Existing and new TUIO servers have to be adapted to be able to correctly communicate with iGesture and we provide a TUIO server specification.

What is TUIO?

TUIO [17] is an open source framework for tangible and multi-touch interaction. Figure 4.4 shows how the TUIO protocol is used for the realisation of a multi-touch table. A camera and a projector are positioned under the table surface. The camera detects and tracks touch input events and the tangible objects. To uniquely identify the objects, the objects are tagged with a unique pattern.

The TUIO tracker or server application sends the tracking information to the client application via the TUIO protocol. The client decodes the messages and handles the events and state changes. Based on the events, the client adapts the user interface that is projected onto the table surface.

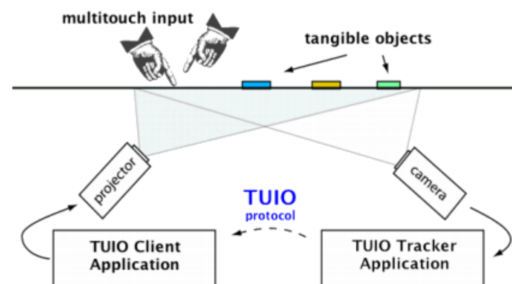


Figure 4.4: TUIO - overview

The TUIO protocol is based on the Open Sound Control (OSC) protocol[10]. OSC is a protocol used for communication between computers and not only musical instruments as the name might suggest. It is optimised for modern networks and can deliver real-time services based on TCP and UDP.

To represent the tangible object, an object profile is available. This profile enables the transmission of information describing an object. The objects can be uniquely identified and their position and rotation can be tracked. For the tracking of touch events, there exists a cursor profile. Cursors do not have a unique identifier and also do not provide any rotation information.

[10] http://opensoundcontrol.org/

In TUIO 1.1, a third profile has been added: the blob profile. The blob profile supports the tracking of objects that cannot be uniquely identified (e.g. objects that were not tagged with a unique pattern). It was introduced to help with the further transition towards TUIO 2.0. All these profiles are defined in a 2D, 2.5D and a 3D version. Furthermore, it is possible to define new custom profiles.

The TUIO protocol specifies four kinds of messages: SET, ALIVE, FSEQ and SOURCE messages. The SET messages are used to transfer information about the state of the objects and cursors and they are sent upon state changes. ALIVE messages communicate the objects that are present on the table surface. An ALIVE message is sent when an object is removed from the surface. Based on the SET and ALIVE messages, the client can deduce the addition and removal of objects. FSEQ messages uniquely identify each update step with a unique frame sequence ID. Each update step consists of a set of SET and ALIVE messages. The SOURCE message is optional and can be used to identify the source of a message. Which is useful if multiple servers send information to the same client. The TUIO message format is shown in Listing 4.9.

Listing 4.9: TUIO message format

```
/tuio/[profileName] set sessionID [parameterList]
/tuio/[profileName] alive [list of active sessionIDs]
/tuio/[profileName] fseq (int32)
/tuio/[profileName] source application@address
```

Integrated TUIO Client

In order to support the use of the TUIO protocol and TUIO devices, a TUIO client has been integrated into the iGesture framework. The implementation is based on the reference implementation of Martin Kaltenbrunner [17], which can be found on the official TUIO website[11]. Currently only version 1.0 of the TUIO protocol is supported.

The structure of the original reference implementation has been modified to make it more flexible in terms of extensions and adaptations. First of all, the handling of messages has been removed from the TUIO client and is now done by separate handlers. For each kind of profile a handler has been defined. A client now only manages the connection, configures what profiles the connection has to listen for and generates the gesture samples. The `TuioConnection` listens for TUIO messages of the configured profiles on a specific port and delegates them to the handlers. The handlers generate add, remove and update events for the objects and cursors. Based on these events, the client generates gesture samples.

Because the implementation relies on Java reflection, new types of messages can be supported by writing a new handler and adding the new message to the profiles file and to the `TuioConstants` class. In the profiles file (an example is shown in Listing 4.10) the mapping between the messages and the handlers is defined. Note that the client has to be updated as well to generate gesture samples based on these new profiles.

[11] http://www.tuio.org/?software

Listing 4.10: tuioprofiles.xml

```
<tuio>
  <association>
    <profile>/tuio/2Dobj</profile>
    <handler>
    org.ximtec.igesture.io.tuio.tuio2D.handler.TuioObjectHandler
    </handler>
  </association>
  <association>
    <profile>/tuio/2Dcur</profile>
    <handler>
    org.ximtec.igesture.io.tuio.tuio2D.handler.TuioCursorHandler
    </handler>
  </association>
  ...
</tuio>
```

The handler can extend the `AbstractTuioObjectHandler` or the `AbstractTuioCursorHandler`. In some cases extra classes must be added. For example, to support the 3D profiles, 3D versions of the `TuioObject`, `TuioCursor` and other relevant classes were created. Because of this flexibility, it is easy to support new custom profiles as well.

Secondly, the user only has to work with a single object/class. For 2D gestures, this is the `TuioReader2D` class whereas for 3D gestures it is the `TuioReader3D` class. The name `TuioReader` refers to either of these classes. The client or `TuioReader` is responsible for setting up the TUIO connection, handling the processed TUIO messages and converting them into 2D or 3D gestures, respectively.

Although the TUIO protocol supports source messages, they have not been implemented in the TUIO client since the specification of the sources within the handlers would make the handlers less generic and less flexible. As a consequence, we suppose that one device or TUIO server corresponds to one `TuioReader`. For each device a separate `TuioReader` is used and every `TuioReader` will listen to a different port.

TUIO messages are transported over a network connection which complicates discovering TUIO devices with the existing discovery service. It is not possible to check all ports to see whether there is a TUIO device associated with that port. This would take too much time and be very inefficient. To solve this issue, a port range is defined in the properties file and only these ports are going to be checked. Furthermore, the TUIO devices do not announce themselves. Another file is used to identify the registered TUIO services as highlighted in Listing 4.11. For each device a port, the device name and the supported gesture types are specified. The port also serves as the unique identifier for the TUIO devices. A 2D TUIO discovery service will instantiate a `TuioReader2D` for each device that supports 2D gestures, a 3D discovery service a `TuioReader3D` for devices that support 3D gestures and so on. For more information about the TUIO implementation classes, the UML diagrams in Appendix A.2 should be consulted.

Because of the inherent differences between the objects and the cursors, the way the gestures are defined are different as well. These differences influence the conversion of the TUIO messages to gestures. The ability to use cursors to perform gestures can be supported very easily. However, in order to support the use of the `TuioObject`s to perform gestures, some changes had to be made to the server as discussed in the next section.

Listing 4.11: tuiodevices.xml

```
<devices>
  <device>
    <port>3333</port>
    <name>TuioSimulator</name>
    <type>2D,3D</type>
  </device>
</devices>
```

Server-side Specification

On a multi-touch table, a `TuioCursor` is represented by a user's finger. The beginning and end of a gesture are clear when using a single finger. The gesture begins when the finger touches the table and the gesture ends when the finger leaves the table. Multi-finger gestures are interpreted as multiple gestures (e.g. a two-finger gesture generates two simple gestures). By defining a composite gesture, a two-finger gesture can be interpreted as a single gesture.

For a `TuioObject`, it is not so straightforward to identify when a gesture starts and when it ends. An object can just lie on the table or it can be pushed over the table to move it. All these actions are not meant to represent gestures. Removing and putting the object on the table again to signal the beginning and end of a gesture is also not a desired behaviour. To avoid this problem, a gesture trigger must be defined. When this trigger is activated, the gesture starts and when the trigger is deactivated, the gesture ends.

Only while the trigger is active, object data must be sent to the client. All other object data is irrelevant since it has nothing to do with gestures. But the object can of course already be on the table before it is used to perform a gesture. Since no object data is sent to the client before the trigger is activated, the client does not know the start position. Therefore, a virtual "add" message for all objects present has to be sent to the client. When the trigger is deactivated, a virtual "remove" message is sent as well. In this way, the client knows when a gesture has finished and the processing can begin. However, these virtual messages may not always arrive at the client via the UDP connection. Therefore, the server may send the same message multiple times. Receiving the same virtual message multiple times does not influence the clients behaviour.

To be compatible with the iGesture framework, the server has to misuse the TUIO protocol by sending virtual add and remove messages. This is of course an unwanted behaviour for other TUIO clients. A second trigger is needed to specify if the server should work in the compatibility or normal mode. In compatibility mode, only gesture data and the virtual add and remove messages are sent. In normal mode, all data is sent but there are no additional add and remove messages.

An extra layer should be defined which the server must use to show the correct behaviour. Since the server can be implemented in any programming language, we were not able to provide this layer for all those languages. Considering that iGesture is programmed in Java, it is possible to do it in Java. However, there are some things that complicate this. First of all, there is no specification about how to construct a server. Adding an extra layer would force a reimplementation of most servers and complicate to use of iGesture.

Secondly, there exist several libraries that provide support for the Open Sound Control protocol, including Illposed Java OSC[12] and NetUtil OSC[13]. The latter has the most features and is more recent.

[12] http://www.illposed.com/software/javaosc.html
[13] http://www.sciss.de/netutil/

Both libraries have their own methods and object types to send OSC messages. It is therefore even difficult to create such a layer for Java implementations only and an interface will be specified.

To indicate the beginning and end of a gesture performed with a tangible object, a gesture trigger (preferably a boolean) should be defined. The corresponding getter and setter methods should be created as well and a trigger to choose between the compatibility mode and the normal mode is required. These triggers should be used to ensure that in compatibility mode object data is only sent during gestures, while in normal mode all data is sent.

Only two more methods have to be defined: `sendVirtualAdd()` and `sendVirtualRemove()`. The `SendVirtualAdd()` method sends a virtual add message for all objects present on the table at a given time. This corresponds to sending an alive message with all objects' session IDs. In this way, the client knows about the objects and gestures can be created. The `sendVirtualRemove()` method sends a virtual remove message for all present objects (by sending an empty alive message). In compatibility mode, the `sendVirtualAdd()` and `sendVirtualRemove()` methods are only called from within the setter of the gesture trigger after setting the trigger value as shown in Listing 4.12.

Listing 4.12: Tuio Pseudo Code 2

```
public void setGestureTrigger(boolean gestureTrigger)
{
        this.gestureTrigger = gestureTrigger;
        if(isCompabilityModeActivated()) {
           if(isGestureTriggered()){
              sendVirtualAdd();
           } else {
              sendVirtualRemove(); }
        }
}
```

4.2.3 Changes to the iGesture Workbench GUI

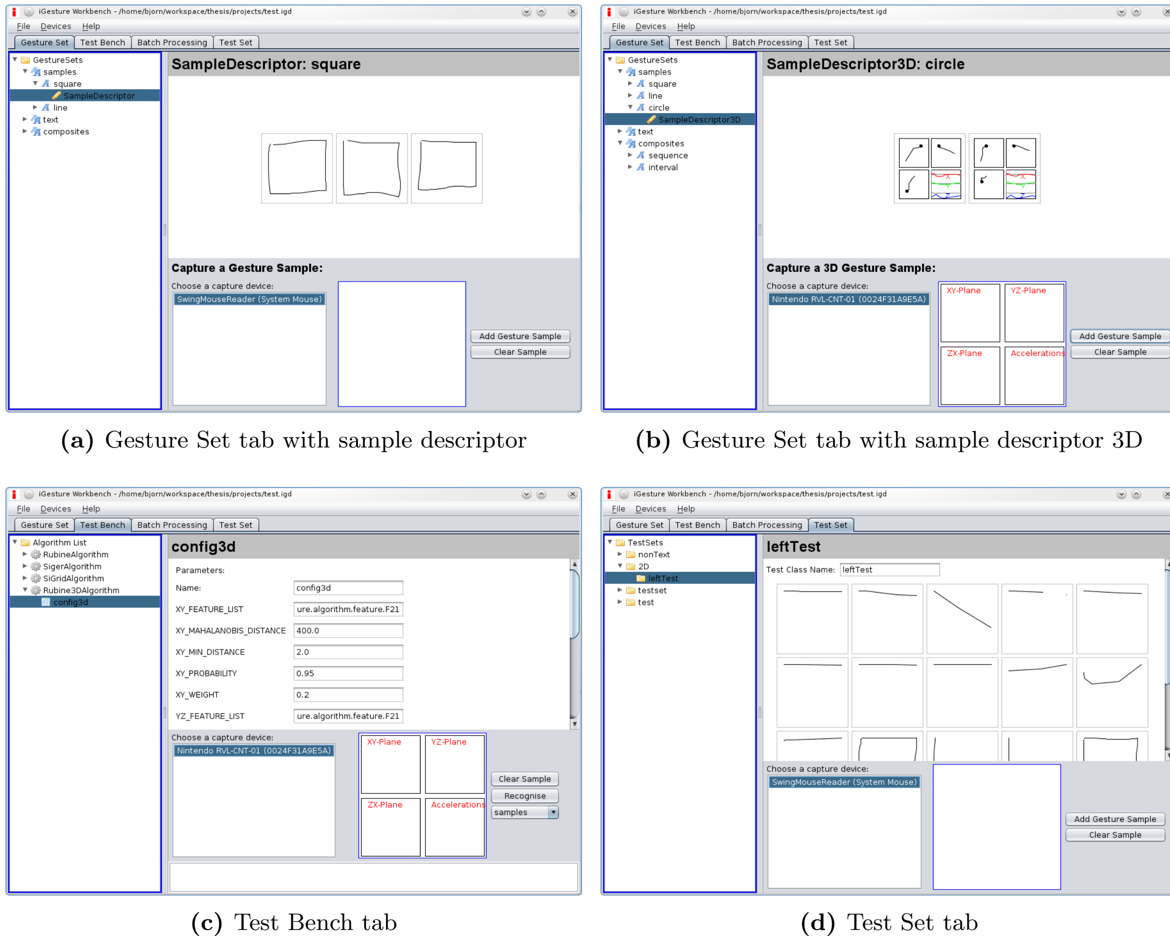
The integration of the device manager and new gesture devices—in particular 3D gesture devices—required some changes to the Graphical User Interface (GUI) of the iGesture Workbench since the Workbench was mainly dealing with 2D gestures and the mouse and digital pens as input devices.

The support for 3D gesture devices required the possibility to add a `SampleDescriptor3D` to a `GestureClass` and to visualise the captured gesture samples as shown in Figure 4.5b. Furthermore, the Rubine 3D algorithm was added to the Test Set tab (see Figure 4.5c) in order to be able to manually test the algorithm's recognition rate.

The old Workbench shown earlier in Figure 2.26, did not allow to select the input capture device without recompilation. The integration of the device manager makes it possible to show a dynamic list of available gesture devices as highlighted in Figure 4.5. When a device is added or removed from the device manager, the list is synchronised. For the `SampleDescriptor` and 2D algorithms, only 2D gesture devices are shown and for the `SampleDescriptor3D` and 3D algorithms only 3D gesture devices. In the Test Set tab all available devices are shown. When a device is selected, the corresponding visualisation panel is shown and the button actions are linked to the selected device.

Some internal changes were needed to accomplish this behaviour, for example the `Algorithm` interface was extended with the `getType()` method returning the type of the algorithm (e.g. 2D, 3D or voice). Several references to the `Note` class were replaced by references to the `Gesture<?>`

interface in order to also support `Note3D` instances. The `Gesture<?>` interface is implementd by the `GestureSample` and `GestureSample3D` classes that encapsulate a `Note` and `Note3D` respectively.



(a) Gesture Set tab with sample descriptor

(b) Gesture Set tab with sample descriptor 3D

(c) Test Bench tab

(d) Test Set tab

Figure 4.5: iGesture Workbench GUI

4.3 Multi-modal Composite Gestures

One of the issues when designing multi-modal gestures is how to define these gestures and how to save them. We introduce an XML format together with the corresponding XML Schema. This schema enables the conversion of different formats as used by third-party tools. Furthermore, the XML Schema can be used to check whether any third-party XML document is correct or not. The manual editing of files may lead to errors which can easily be detected by validating against the XML Schema. To define the multi-modal gestures, a graphical user interface which is introduced in the second part of this section, can be used.

4.3.1 XML Schema

The iGesture framework supports the recognition of simple gestures and based on the work presented in this thesis also composite gestures. Of course, there is a need to persistently store these gestures. Therefore, an XML format has been developed. To make conversion between existing formats and the format used by iGesture easier, an XML Schema was defined, which can be found in Appendix B.

A simple gesture can be either described by a textual descriptor or by one or more gesture samples. Listing 4.13 represents a square gesture that has been described using a textual descriptor. With a textual descriptor, a gesture can for example be represented as a combination of directions (East (E), South (S), West (W) and North (N)).

Listing 4.13: Text Descriptor

```
<igs:class name="square" id="fbca38d2-3375-4015-8acd-dda571b05858">
  <igs:descriptor type="org.ximtec.igesture.core.TextDescriptor"
   id="6b18d0a5-1c79-49a2-83e0-2a6d5b77476f" xsi:type="igs:TextDescriptorType">
    <igs:text>E,S,W,N</igs:text>
  </igs:descriptor>
</igs:class>
```

The iGesture namespace is defined by with the prefix `igs`. Each *class* and *descriptor* element is identified with a universally unique identifier (UUID). The *class* element represents a `GestureClass` and each `GestureClass` contains a `Descriptor` represented by the *descriptor* element of type DescriptorType. The DescriptorType is a generic type which has to be derived for example by TextDescriptorType. In order to use an instance of TextDescriptorType everywhere an instance of DescriptorType is expected, the intended derived type must be identified. The derived type is identified by using the *xsi:type* attribute which is part of the XML Schema instance namespace.

A gesture can also be described by samples as mentioned earlier. Listing 4.14 shows an example of a circle gesture. The sample descriptor can contain several samples for the same gesture. Each sample is a note consisting of a number of traces. A trace is a sequence of points. In the example, the points are 2D points. Each point has x and y coordinates and a timestamp. A similar format is also provided for 3D points. In addition to the point data, the acceleration data can be saved as well for 3D gestures. Just like the *descriptor* element, the *point* element is a generic element that has to be extended. Often, the use of simple gestures is sufficient. In some cases, it can be useful to combine several gestures into more complex gestures or to combine gestures made by different devices and/or users. For this type of complex gestures we can define timing and distance constraints. We identified six kinds of gesture combinations:

1. concurrent: concurrent gestures
2. sequence: a sequence of gestures
3. proximity and concurrent: concurrent gestures that are performed close to each other
4. proximity and sequence: a sequence of gestures that are performed close to each other
5. interval : multiple gestures that are performed within a given time interval
6. cardinality : within a time interval, a particular gesture must be performed between a minimum and a maximum number of times. Each gesture must be performed by a different device or user.

Time, distance and the number of times a gesture has to be performed are not the only parameters that can be used to define a constraint. It is furthermore possible to specify for each gesture forming part of one composition by which user and/or by which kind of device it has to be captured. In the latter case, it is also possible to exactly specify which devices are allowed. For example, two gestures have to be performed concurrently by different users. Or all users should use a device of the same type to perform a cardinality gesture.

Listing 4.14: Sample 2D and 3D descriptor

```
<igs:class name="circle" id="ebca38d2-3375-4015-8acd-dda571b05858">
  <igs:descriptor type="org.ximtec.igesture.core.SampleDescriptor"
                           id="7b18d0a5-1c79-49a2-83e0-2a6d5b77476f"
                           xsi:type="igs:SampleDescriptorType">
    <igs:sample name="" id="095d26d7-842a-4317-899c-c92c78a3258c">
      <igs:note>
        <igs:trace>
            <igs:point xsi:type="igs:Point2DType">
              <igs:timestamp>2010-02-08T12:55:26.000</igs:timestamp>
              <igs:x>45.0</igs:x>
              <igs:y>90.0</igs:y>
            </igs:point>
            <igs:point xsi:type="igs:Point2DType">
              <igs:timestamp>2010-02-08T12:55:26.000</igs:timestamp>
              <igs:x>46.0</igs:x>
              <igs:y>90.0</igs:y>
            </igs:point>
            ...
          </igs:trace>
      </igs:note>
    </igs:sample>
    ...
  </igs:descriptor>
</igs:class>
<igs:class name="square" id="gbca38d2-3375-4015-8acd-dda571b05858">
  <igs:descriptor type="org.ximtec.igesture.core.SampleDescriptor3D"
                           id="5b18d0a5-1c79-49a2-83e0-2a6d5b77476f"
                           xsi:type="igs:SampleDescriptor3DType">
    <igs:sample name="" id="295d26d7-842a-4317-899c-c92c78a3258c">
      <igs:note3D>
        <igs:point3D xsi:type="igs:Point3DType">
          <igs:timestamp>2010-02-08T12:55:26.000</igs:timestamp>
          <igs:x>45.0</igs:x>
          <igs:y>90.0</igs:y>
          <igs:z>45.0</igs:z>
        </igs:point3D>
        ...
        <igs:acceleration>
          <igs:sample>
            <igs:timestamp>2010-02-08T12:55:26.000</igs:timestamp>
            <igs:xAcc>45.0</igs:xAcc>
            <igs:yAcc>90.0</igs:yAcc>
            <igs:zAcc>45.0</igs:zAcc>
          </igs:sample>
          ...
        </igs:acceleration>
      </igs:note3D>
    </igs:sample>
  </igs:descriptor>
</igs:class>
```

Listing 4.15 shows the definition of a composite gesture. The *constraint*, *param* and *descriptor* element are again generic elements to support simple extensibility. A constraint consists of an enumeration of the gestures that are part of the composite gesture and the parameters of the constraint. Here, a proximity and sequence constraint is used to define a composite gesture consisting of two gestures. Each gesture is represented by a *gesture* element. In this case the last gesture has to be performed by user 0 and with a `WiiReader`. The user is identified by a number which is mapped to a real user at runtime. For the other gestures, it does not matter who performs it since the optional *user* attribute is missing. Note that the *device* attribute is optional except for proximity-based constraints.

Listing 4.15: Composite descriptor

```
<igs:class name="composite" id="hbca38d2-3375-4015-8acd-dda571b05858">
  <igs:descriptor type="org.ximtec.igesture.core.CompositeDescriptor"
                   id="4b18d0a5-1c79-49a2-83e0-2a6d5b77476f"
                   xsi:type="igs:CompositeDescriptorType">
    <igs:constraint id="d167bea9-72bb-454d-a09f-39a491239e6b"
                   type="proximitysequence"
                   xsi:type="igs:ProximitySequenceConstraintType">
      <igs:gesture id="1" idref="ebca38d2-3375-4015-8acd-dda571b05858"
                   device="WiiReader"/>
      <igs:gesture id="2" idref="gbca38d2-3375-4015-8acd-dda571b05858"
                   user="0" device="WiiReader"/>
      <igs:param>
        <igs:minTime>00:00:00.000</igs:minTime>
        <igs:maxTime>00:01:00.000</igs:maxTime>
        <igs:minDistance>0.5</igs:minDistance>
        <igs:maxDistance>1.0</igs:maxDistance>
        <igs:distanceUnit>m</igs:distanceUnit>
      </igs:param>
    </igs:constraint>
  </igs:descriptor>
</igs:class>
```

The *param* element describes the parameters of the constraint. First of all, there is an allowed time frame between two consecutive gestures of minimal zero seconds and maximal one minute. The diagonal of the combined bounding box of the gestures that form part of the composite, may not be shorter than half a meter and not longer than one meter. The *idref* attributes are generally used to refer to other elements and here specifically to other gestures. For example, the *idref* attributes of both *gesture* elements refer to the circle and square gesture from Listing 4.14 respectively. It is possible to define a complex gesture in multiple steps. This is possible since the *idref* attribute of an *gesture* element can refer to any simple or complex gesture. The *id* attributes are used to uniquely identify elements. For the *gesture* elements, the *id* element is only unique within the surrounding *constraint* element.

An extended version of the composite descriptor exists as well which enables the specification of different parameters for every gap between two consecutive *gesture* elements as shown in Listing 4.16. In the example in Listing 4.16, a sequence of three gestures has to be carried out. Like in the previous example, the last gesture has to be performed by user 0 and with a Wii Remote. But in this case, the devices element is used to specify exactly which Wii Remotes (identified by MAC address) may be used to perform the gesture.

The first *param* element is used to indicate that the timeframe between the first and second gesture is between 30 and 45 seconds. While the second *param* element indicates that for all other gaps a time frame between 10 and 15 seconds is used. The string “default” is used to signal that these parameters are valid for all gaps that have no specific parameters assigned. Note that it is also possible to specify a list of ids instead of just a single one. The *idref* element within a *param* element or a *devices* element refers to a *gesture* element within the same surrounding *constraint* element.

Listing 4.16: Extended composite descriptor

```
<igs:class name="xcomposite" id="ibca38d2-3375-4015-8acd-dda571b05858">
   <igs:descriptor type="org.ximtec.igesture.core.CompositeDescriptor"
                                 id="3b18d0a5-1c79-49a2-83e0-2a6d5b77476f"
                                 xsi:type="igs:CompositeDescriptorType">
      <igs:constraint id="e167bea9-72bb-454d-a09f-39a491239e6b"
                                 type="xsequence"
                                 xsi:type="igs:XSequenceConstraintType">
         <igs:gesture id="1" idref="ebca38d2-3375-4015-8acd-dda571b05858" />
         <igs:gesture id="2" idref="fbca38d2-3375-4015-8acd-dda571b05858" />
         <igs:gesture id="3" idref="gbca38d2-3375-4015-8acd-dda571b05858"
                                 user="0" device="WiiReader"/>
         <igs:devices>
            <igs:devicename>00:11:22:33:44:55</igs:devicename>
            <igs:devicename>AA:BB:CC:DD:EE:FF</igs:devicename>
            <igs:devicename>00:BB:22:DD:44:FF</igs:devicename>
            <igs:idref>3</igs:idref>
         </igs:devices>
         <igs:param>
            <igs:minTime>00:00:30.000</igs:minTime>
            <igs:maxTime>00:00:45.000</igs:maxTime>
            <igs:idref>1</igs:idref>
         </igs:param>
         <igs:param>
            <igs:minTime>00:00:10.000</igs:minTime>
            <igs:maxTime>00:00:15.000</igs:maxTime>
            <igs:idref>default</igs:idref>
         </igs:param>
      </igs:constraint>
   </igs:descriptor>
</igs:class>
```

Finally, the gestures have to be assigned to a gesture set as shown in Listing 4.17.

Listing 4.17: Gesture set definition

```
<igs:set name="MyGestures" id="b21e7eb9-7e7b-40db-9d60-8b9212576237">
    <igs:class idref="ebca38d2-3375-4015-8acd-dda571b05858"/>
    <igs:class idref="fbca38d2-3375-4015-8acd-dda571b05858"/>
    <igs:class idref="gbca38d2-3375-4015-8acd-dda571b05858"/>
    <igs:class idref="hbca38d2-3375-4015-8acd-dda571b05858"/>
    <igs:class idref="ibca38d2-3375-4015-8acd-dda571b05858"/>
</igs:set>
```

4.3.2 Composite Descriptor GUI

In the previous section, the different constraints and descriptors of composite gestures were defined. The way they are stored has also been defined through an XML Schema. Of course, there is a need for an alternative way to define the gestures rather than to manually describe them in an XML document.

A first possibility that was considered was the definition a new syntax to specify and edit the multi-modal gestures. A parser for this syntax can be easily created with tools like ANTLR[14]. Only a description of the syntax in Backus-Naur-Formalism (BNF) is needed. However, there would not be a significant difference from defining the gestures in XML and an XML editor could be used in the GUI as well. The correctness of a new description could be validated against the defined XML Schema.

A more user-friendly way to define multi-modal gestures is needed. It became clear soon that there exist tools that can generate GUIs based on an XML Schema specification. The Eclipse Modelling Framework (EMF)[15] and Graphical Modelling Framework (GMF)[16] are such tools. The problem is that these tools are very tightly coupled to the Eclipse platform and need an Eclipse Environment to run. As a consequence of using these tools, the iGesture framework would become much bigger and heavier to use. After some research on the Internet, a library which provides similar functionality was found. JaxFront[17] can be used to dynamically generate GUIs at runtime based on an XML Schema. The library is available under an open-source and commercial license. Either way, a license fee has to be paid and we decided not to use JaxFront.

The last idea to define and edit the gestures in a graphical way was to create a GUI based on the JGraph[18] library. Using this library, diagram and graph editors can be created. However, implementing a JGraph-based GUI requires quite some time and this task therefore had to be postponed for future work.

Finally, a GUI was designed that supports the manual definition and editing of the composite gestures and makes it possible to change their characteristics as shown in Figure 4.6. On the left-hand side, the defined gesture sets, gesture classes and descriptors are shown. If the user adds a composite descriptor to a gesture class, a dialogue is shown where the user can select a constraint type from a combo box. These constraint types are read from the workbench configuration which implies that the addition of a new constraint type does not affect the source code of the GUI. The top half of the tab shows the constraint parameters. In the example we see a sequence constraint and the parameters are the minimum and maximum time between each two consecutive gestures. The bottom half of the tab is used to add gestures to the composite. After selecting a gesture set, a gesture class from that set can be selected. If a gesture class has been selected, it can be added to the composition. By ticking the user check box, it is possible to select a user that should perform the gesture. By ticking the device check box, it becomes possible to select the kind of device that should be used to perform the gesture. The available device types are also read from the workbench configuration. Once a device type has been selected, the identifiers of all devices of that type that are connected to the workbench are shown in the device list and specific devices can be selected.

When a gesture has been added to a composite gesture, it is shown in the list at the bottom of the window. Any gesture can be removed from the composite by selecting it and pressing the remove button and all gestures can be removed from the composite by clicking the clear button.

[14]http://www.antlr.org/
[15]http://www.eclipse.org/modeling/emf/
[16]http://www.eclipse.org/modeling/gmf/
[17]http://www.jaxfront.com/
[18]http://www.jgraph.com/jgraph.html

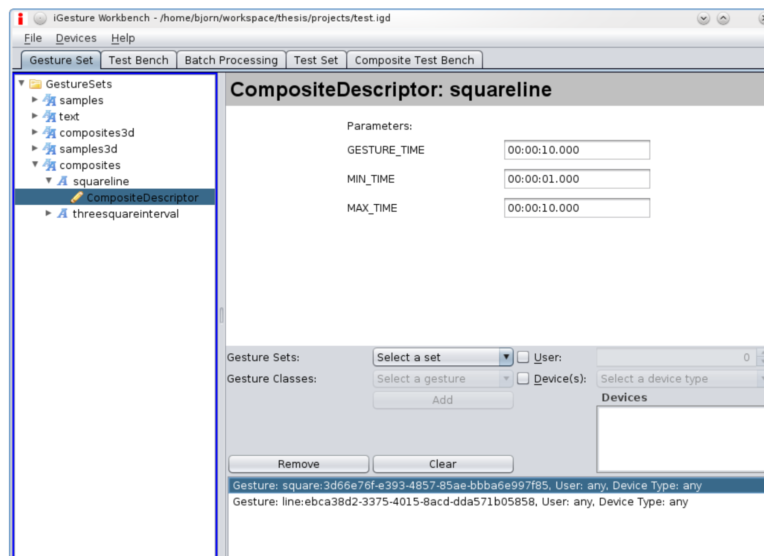


Figure 4.6: Gesture Set tab with CompositeDescriptor GUI

As part of future work, a component could be created that enables the definition of composite gestures by actually performing them. A JGraph-based GUI could then be used to further edit these composite gestures.

4.4 Implementation of a Multi-modal Recogniser

In Chapter 3, we introduced the basic concepts of our multi-modal recogniser. The implementation of these concepts is discussed in this section. We begin by introducing the components that form part of the multi-modal recognition architecture, followed by the algorithm used by the multi-modal recogniser and discuss how it influences the implementation of the constraints. Then the implementation of the constraints themselves is introduced and we conclude with the description of a GUI for the manual testing of composite gestures.

4.4.1 Multi-modal Recognition Architecture Components

In Chapter 3, three components and their behaviour have been introduced: the multi-modal manager, the multi-modal recogniser and the gesture queue. The `MultimodalGestureManager` class implements the multi-modal manager, the `MultimodalGestureRecogniser` class implements the multi-modal recogniser and the queue is implemented by the `MultimodalGestureQueue` class.

Some changes were made to the iGesture framework to support the behaviour described in Chapter 3. The multi-modal manager sends recognition results to the `GestureHandlers` registered with the `Recogniser` producing these results. In order to do this, the source must be known and therefore, a source attribute had to be added to the `ResultSet` class representing the recognition results. A source field was added to the `Gesture` interface as well in order to know which device and what kind of device created the gesture sample.

The `ResultSet` encapsulates the recognised gesture sample without providing the name of a specific gesture. The `Recogniser` sets the name of the gesture sample to the `GestureClass` with the highest probability. Depending on the `GestureClass` of the gesture sample, the multi-modal manager decides to either push it in the queue or to send it to the handlers registered with the source `Recogniser`.

4.4.2 Multi-modal Recognition Algorithm

The algorithm was introduced in Section 3.3 and visualised in Figure 3.4. The first phase consists of creating the character representation. In this phase, the `MultimodalGestureRecogniser` first asks all the composite gestures in the set it has been configured with, what their composing gestures are. When all the different gesture classes that form part of a composite gesture are known, a single character representation is chosen for each gesture class. The character can be any textual Unicode character which means that there can be $\pm\ 2^{16}$ possible different gestures that can be used to form a composite. The first character that is used is the "#" character with a decimal value of 35. The other characters are allocated in increasing order.

The next phase is to generate the patterns for all composite gestures based on the character representation generated in the previous phase. Since the number of patterns that can represent a composite gesture differs based on the used constraint and because new constraints may be added in the future, we decided to not hard code the pattern generation in the algorithm itself. In this way the algorithm remains general and does not have to be changed when a new constraint type is added. The pattern generation is done by the constraint itself. Once all patterns are known, the actual recognition process can begin. Note that the pattern length is limited to the word length of the machine (typically 32) since the algorithm uses bitwise comparison techniques.

Based on the time windows, a part of the queue is copied and its string representation is generated. The string representation is also created based on the character mapping defined in phase 1 and serves as the input for the next phase. The time windows are also generated during the initialisation of the `MultimodalGestureRecogniser`. Once all gestures are known that form part of a composite, the multi-modal recogniser asks the constraints for the time window for all the gestures that are part of them. For each composing gesture, the maximum time window is used.

In the next phase, the multi-modal recogniser looks for a match of a pattern in the string representation of the queue. The Bitap algorithm is used to perform the pattern matching. An implementation of this algorithm was found in the google-diff-match-patch[19] project under an Apache 2.0 license. Since the number of patterns may be large, the pattern matching is performed in parallel by multiple threads. Each thread has its own set of patterns (the default size is 10).

If a potential match has been found, the match and the pattern are compared to validate the output of the Bitap algorithm. Based on the differences between the two strings, the indexes of the gestures that are needed to check the additional constraints are determined. If something was deleted from the text to form the pattern, that something is not needed and the index can be skipped. If something is equal, the index is put in the list of indexes. If something was inserted, it means that there is something missing in the queue and there is no match. The following example explains the mechanism.

text	d	a	c		where D is deleted
pattern		a	b	c	where E is equal
	D	E	I	E	where I is inserted

If the text "dac" is compared with the pattern "abc", then in the pattern the character "d" was deleted, the character "a" is equal, the character "b" was inserted and the character "c" is equal as well. According to the diff process, b was inserted in the text to form the pattern. This means that gesture b is not present in the queue and therefore not all required gestures are available. As a

[19] http://code.google.com/p/google-diff-match-patch/

consequence, there is no valid match. If there is a valid match and once all indexes are known, the corresponding copied gestures are used to check the constraints. Here we face the same problem as with the pattern generation, where conditions differ from constraint to constraint. Instead of hard coding the condition checks into the algorithm, they are delegated to the constraints themselves.

If all the conditions of a constraint are satisfied, there is a valid match for the composite gesture in the queue and the `GestureHandler`s registered with the `MultimodalGestureRecogniser` are notified of this event. The composing gestures are also marked as part of a validated composite gesture. If there is no valid match, the next pattern is checked. Note that gesture samples that are marked as part of a validated composite are not copied along with the other samples as part of a time window. Every time something is pushed into the queue, all gesture samples within the time window of the last inserted sample are copied from the queue and the patterns are checked.

One last issue still has to be discussed: the cleanup of the queue. A separate thread is used to perform the garbage collection. When the garbage collector thread wakes up, it checks whether the queue contains at least a minimum number of samples. If this is the case, it starts to remove elements from the head of the queue if and only if the sample is not covered anymore by a time window. One by one the elements are removed until the thread encounters a sample that lies still within a time window or until the queue contains the minimum number of samples. After each run the thread is sent asleep for a fixed amount of time. The sleep time and the minimum number of elements that should be present in the queue can be configured. The default values are one minute for the sleep time and a minimum of five elements should be present in the queue.

The garbage collection thread can be configured to check whether a gesture sample was validated as part of a composite gesture. If not, the thread notifies the registered `GestureHandler`s of the source `Recogniser` of the performed gesture. Since the behaviour of the garbage collection thread is unpredictable, there are no guarantees about the delivery time of these gestures that potentially form a composite. Therefore, it is advised not to listen for composite gestures and the gestures that form part of those composites at the same time.

At the moment, we assume that there is only a single-level gesture composition. However, multi-level composites where a composite is composed out of other composites can also be supported. During its initialisation, the `MultimodalGestureRecogniser` can recursively obtain the gestures that compose a gesture. Note that a composite that is part of another composite must belong to the same gesture set as the top-level composite since only the name of the gesture class is referenced in the constraint. This way, the gestures that form part of a the lower level composite can be obtained. If a composite was recognised, a virtual gesture sample could be created that represents the composite and be put in the gesture queue for recognising other composite gestures.

4.4.3 Constraints

Multi-modal composite gestures are described by a `CompositeDescriptor` which is shown in Figure 4.7. Each `CompositeDescriptor` has a constraint which implements the `Constraint` interface shown in Listing 4.18.

New constraints can be added by implementing the `Constraint` interface. We provide a basic set of constraints that represent commonly used composite gestures. These constraints can also be extended.

Listing 4.18: Constraint Interface

```
public interface Constraint {

    public void addGestureClass(String gestureClass) throws
         IllegalArgumentException;
    public void addGestureClass(String gestureClass, int user) throws
         IllegalArgumentException;
    public void addGestureClass(String gestureClass, String deviceType, Set<
         String> devices) throws IllegalArgumentException;
    public void addGestureClass(String gestureClass, int user, String deviceType,
          Set<String> devices) throws IllegalArgumentException;

    public void removeGestureClass(DefaultConstraintEntry entry);
    public void removeAllGestureClasses();

    public List<String> getGestureClasses();
    public Set<String> getDistinctGestureClasses();
    public int getNumberOfGestures();
    public List<DefaultConstraintEntry> getGestureEntries();

    public boolean validateConditions(List<Gesture<?>> gestures, IDeviceManager
         manager);
    public Set<String> generatePatterns(Map<String, String> charMapping);
    public Map<String, Calendar> determineTimeWindows();

    public Map<String, String> getParameters();
    public String getParameter(String property);
    public void setParameter(String property, String value);
    public String toString();
}
```

We have defined the following constraints and a class diagram of these constraints can be found in Appendix A.3:

- `ConcurrencyConstraint`: concurrent gestures
- `SequenceConstraint`: a sequence of gestures
- `IntervalConstraint`: multiple gestures that are performed within a time interval
- `CardinalityDeviceConstraint`: within a time interval, a particular gesture must be performed between a minimum and a maximum number of times. Each gesture must be performed with a different device.
- `CardinalityUserConstraint`: Here each gesture must be performed by a different user.
- `ProximityConcurrencyConstraint`: concurrent gestures that are performed close to each other
- `ProximitySequenceConstraint`: a sequence of gestures that are performed close to each other

A composite gesture is composed by a number of gestures. Therefore, the `Constraint` should keep a record of this. Gestures can be added to the constraint using one of the four overloaded `addGestureClass()` methods. As mentioned earlier, it is possible to specify a user and device for each composing gesture. Since these attributes are optional, the overloaded methods have been introduced. The `addGestureClass()` method can throw an `IllegalArgumentException`. This exception may be thrown by the `CardinalityDeviceConstraint`, the `CardinalityUserConstraint`, the `ProximitySequence-` and `ProximityConcurrencyConstraint`.

A cardinality-based constraint may only contain one gesture class and this gesture has to be performed by different users. If the constraint already contains a gesture class and a new gesture class is added to the constraint, an `IllegalArgumentException` is thrown to indicate that the new gesture class was not added. If the user field is specified when a gesture class is added to a cardinality-based constraint, the user field is ignored. The gesture class is added to the constraint if the constraint does not already contain a gesture class, an exception is still thrown to indicate that the user field was ignored.

For proximity based constraints, it is important that the different gestures can be logically compared. Therefore, the device type (e.g. `WiiReader`, `TuioReader2D`) has to be specified to determine if the gestures can be compared. If they can not be logically compared, an `IllegalArgumentException` is thrown. 2D gestures can only be compared with 2D gestures and 3D gestures with other 3D gestures. For example, a gesture performed with a `TuioReader3D`—a 3D gesture—can not be compared with a gesture created with a `TuioReader2D`—a 2D gesture. Gestures that were performed on a `TuioReader2D` can be compared. Note that it is up to the user to make sure that the devices use the same distance units and use the same origin to determine the coordinates.

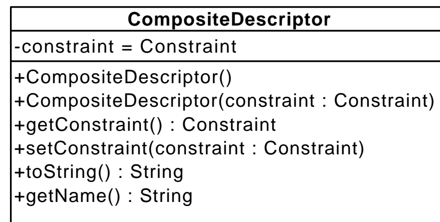


Figure 4.7: CompositeDescriptor class diagram

A `DefaultConstraintEntry` object (see Figure 4.8) encapsulates the gesture class name, the character representation and the optional information about the user, device type and specific devices. To simplify the comparison and look-up of the gesture information during the recognition process, the gesture class name is used instead of a reference to the `GestureClass` object itself. However, this name is not unique. Multiple gesture sets can contain a gesture with the same name. Therefore, the name of the `GestureClass` is concatenated with the Universally Unique Identifier (UUID) of the `GestureClass` object. The UUID of a `GestureClass` instance is written to a file when the gesture sets are saved and reused when the gesture set is loaded again.

A gesture can also be removed from a `Constraint` with the `removeGestureClass()` method. All gestures can be removed from the `Constraint` at once to using the `removeAllGestureClasses()` method. Some getter methods allow the user to obtain the names of the gestures that form part of the composite (e.g. in the character representation generation phase), the number of gestures that compose the multi-modal gesture or to obtain the `DefaultConstraintEntry` objects themselves (e.g. to display the composite gestures in the GUI after loading a project from file).

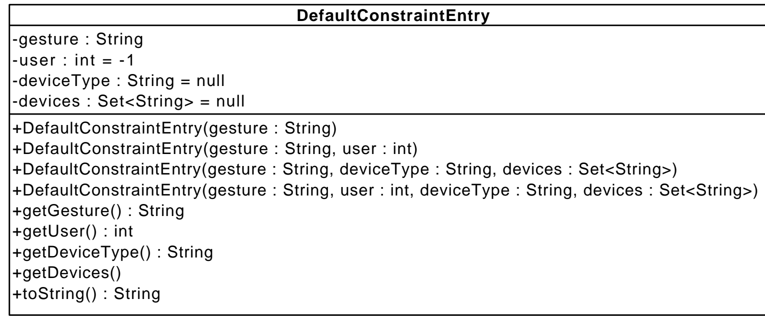


Figure 4.8: DefaultConstraintEntry Class Diagram

Each constraint has one or more parameters (e.g. duration or distance). A getter and setter method is provided to obtain the value of a particular parameter or to change its value. The `getParameters()` method returns a mapping of the names of all properties to their respective values. The remaining three methods were added to the `Constraint` interface to make the multi-modal recognition algorithm more general and flexible. New constraints can be added without having to change the algorithm itself.

The `determineTimeWindows()` method generates the maximal time window for each kind of composing gesture as described in Chapter 3. The patterns that represent the composite gesture are generated by the `generatePatterns()` method. This method takes a mapping from the name of the gesture classes that form part of the composite to the character representation as a parameter.

The `validateConditions()` method validates the conditions of the constraint and takes a list of gesture samples and a reference to the device manager as arguments. The conditions in the defined constraints are either time, space, user or device related. The start and end timestamps of the gesture samples are used to verify whether the samples were performed in parallel, sequentially or within a certain time interval. To check whether two gestures are performed sequentially, the start timestamp of one gesture must come after the end timestamp of the other plus the minimum gap time and before the end timestamp of the other plus the maximum gap time. This comparison is done for each two consecutive gestures.

$$\forall t_{end,i}, t_{start,i+1}, \quad t_{end,i} + t_{gap_min} \leq t_{start,i+1} \leq t_{end,i} + t_{gap_max}$$

To verify whether multiple gestures were carried out within a certain interval, the start timestamp of the first performed gestures is incremented with the interval time and if all other gestures end before that moment, all gestures were performed in that interval.

$$\forall t_{end,i}, \quad t_{end,i} \leq t_{start,1} + t_{interval}$$

To validate if multiple gestures are executed in parallel, the end timestamp of the first performed gesture is compared to the start timestamp of all the other gestures. If all other gestures start before the end timestamp of the first gesture, they all overlap in time and are according to our definition performed in parallel. The different time conditions are visualised in Figure 4.9.

$$\forall t_{start,i}, \quad t_{start,i} \leq t_{end,1}$$

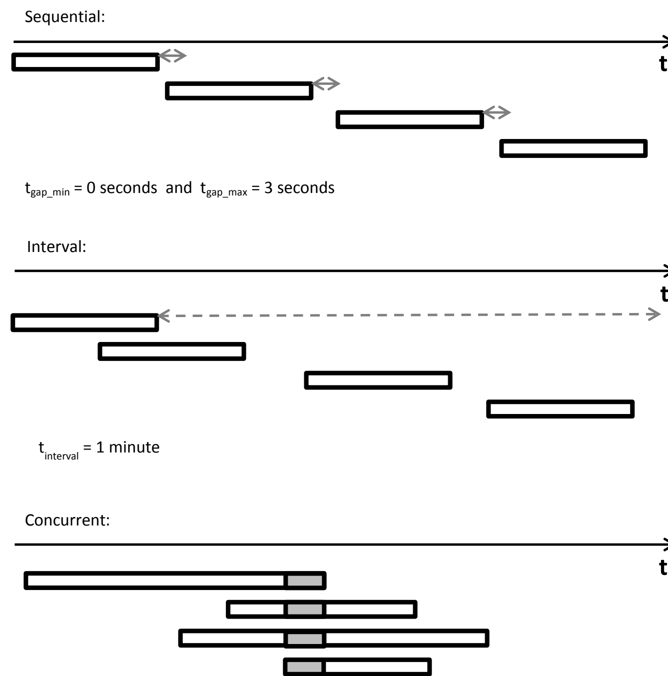


Figure 4.9: Time conditions

The coordinates of the points (`Point` and `Point3D`) of the gesture samples are used to check the space related conditions. A combined 2D or 3D bounding box is calculated. Based on Pythagoras' theorem, the length of the diagonal of the bounding box or the diagonal of the largest face of the 3D rectangular bounding box is calculated. If the length of the diagonal is larger than or equal to the minimum specified length and smaller than or equal to the maximum specified length, the space related conditions are met. The following formula visualises the specified conditions:

$$d_{minimum} \leq d_{actual} \leq d_{maximum}$$

The device conditions can be checked by verifying if the device that was used to generate the gesture sample, is of the correct type of device and has an allowed ID. To check the user conditions, the device manager is needed since only the device manager contains information about which user is associated with a device. The first step is to iterate over the gesture samples and to create a mapping from the `User` to a list of names of gestures that the user performed. Then, a similar mapping is created for the gestures defined in the constraint. Finally, for each defined user, the recogniser checks whether a `User` can be found who performed at least the required matching gestures. An example where for three gestures the user was defined, for the other gestures it does not matter:

Defined users:		
0	"square"	
1	"line"	"circle"
Recognised users:		
Bjorn	"square"	"line"
William	"triangle"	
Beat	"line"	"circle"
Sven	"diagonal"	

The recogniser maps Bjorn to user 0 and Beat to user 1. Each defined user was matched with a runtime user and as a consequence, the user conditions are correctly validated. The first found shortest match is used, so if user Sven performs a “square” instead of a “diagonal”, then user Sven will be mapped to user 0.

For a `CardinalityUserConstraint`, the user who performed the gesture is not specified since the semantics of the constraint implies that each gesture is performed by a different user and a user is allowed to give only one vote! Therefore, the user condition cannot be checked in the same way as for other gestures. Instead, all gesture samples are first iterated and the users who performed the gestures are put in a set. If the number of users in the set is equal to the number of gesture samples, the condition holds. A similar approach is used to check the condition that each device has only one vote in the case of a `CardinalityDeviceConstraint`, even if a user can use multiple devices. As soon as one of the conditions is not satisfied, the validation process is stopped.

Because the multi-modal recogniser contains multiple threads running in parallel, it is possible that certain side effects might arise if, for example, a concurrent constraint for two “triangle” gestures and an interval constraint for three “triangle” gestures have been defined. If two “triangle” gestures were performed in parallel as part of the interval constraint, it is possible that the multi-modal recogniser recognises these two gestures as the concurrent constraint before the third “triangle” gesture is performed leading to undesired behaviour. The concurrent constraint can be seen as part of the interval constraint and as we mentioned earlier it is not advised to listen at the same time for composite gestures and the gestures—simple or composite—that form part of them. Other issues may arise from race conditions, and the inspection of parallel programs is not easy. Therefore, a visualisation component should be created as part of future work to visualise the behaviour of the queue and facilitate potential debugging.

We can now create and edit the descriptors and constraints, but we need to be able to store them as well. In Section 2.3, we mentioned that the gesture sets, classes and descriptors are saved to either an XML format or to a db4objects database. In order to store an object to a db4objects database, the object's class only has to extend the `DefaultDataObject` class.

The conversion from the objects to the XML format and back is executed using JDOM[20]. Each class has a corresponding JDOM class, for example, the `JdomCompositeDescriptor` class corresponds to the `CompositeDescriptor` class. A JDOM class has a constructor which creates the XML representation of the object, and an `unmarshal()` method which creates an object from the corresponding XML elements. The JDOM classes for the constraints follow the XML Schema that was defined in Section 4.3.1. The other classes may not follow the XML Schema to support the old files during a transition period.

4.4.4 Composite Test Bench

In Section 4.3.2, we introduced a GUI to define and edit the composite and multi-modal gestures. Now we will describe a GUI that can be used to manually test the recognition of composite gestures. A new tab was created to offer this functionality as shown in Figures 4.10 and 4.11. The Composite Test Bench tab is for composite gestures what the Test Bench tab is for simple gestures. The multi-modal recognition architecture shown in Figure 3.1 is configured and managed by this tab in two steps. The first step is to configure the `Recogniser`s and devices, the second step is configuring the multi-modal manager and multi-modal recogniser.

[20]http://jdom.org/

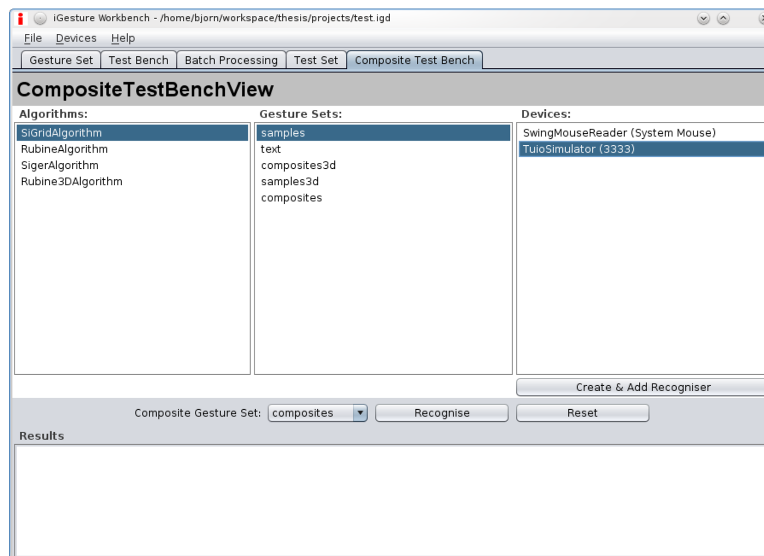


Figure 4.10: Composite Test Bench configuration

The upper half of the tab is used to configure the different devices and the `Recogniser`s. The user can select one or more algorithms and gesture sets to configure a `Recogniser`. The selected devices send the gesture input to that `Recogniser`. Multiple `Recogniser`s can be created and are added to a list.

The lower half of the tab controls the multi-modal gesture manager and recogniser. The user can select the gesture set to configure the multi-modal recogniser. If the set does not contain any composite gestures, the user is shown an error message and is asked to select a different gesture set. Then the multi-modal manager is configured with the multi-modal recogniser and the `Recogniser`s that were created in the previous step.

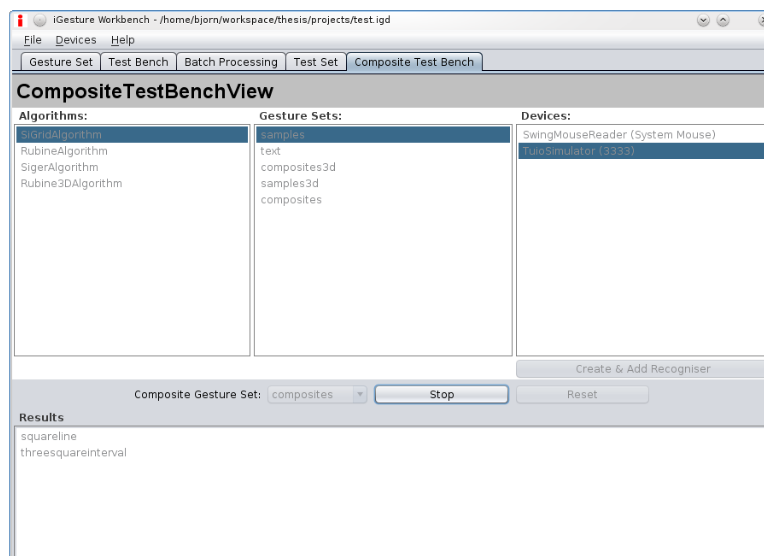


Figure 4.11: Composite Test Bench recognition

By pressing the *Recognise* button the multi-modal recognition process is started and is stopped by pressing it a second time. The recognition results are shown in the list at the bottom of the window. The *Reset* button resets the whole configuration and GUI. The user can now create other `Recogniser` configurations and reconfigure the multi-modal recogniser. In this example, a single `Recogniser` was configured with the SiGrid algorithm and the samples gesture set. A TUIO device on port 3333 sends the gesture samples to the `Recogniser` for recognition which on its turn sends the recognition results to the multi-modal manager since it is in multi-modal mode. The multi-modal recogniser is configured with the composites gesture set. During the recognition process, the configurations cannot be changed. The result list shows that a “squareline” and a “threesquareinterval” gesture were recognised.

Chapter 5

Applications

In this chapter, we introduce some applications that could benefit from multi-modal gesture interaction. The first example is a multimedia player. Another application is the PaperPoint presentation tool. Both applications are used in a multi-user context. The last example is Geco, an application launcher.

5.1 Multimedia Player

Media center applications such as Windows Media Center[1], XBMC[2], Boxee[3] or MythTV[4] are gaining in popularity and offer a lot of features. In most cases, these applications are controlled with a remote control everyone is used to from watching TV. However, accessing all these features can be cumbersome. The user has to press multiple buttons or has to walk through several steps to activate or use a particular feature. Some remotes offer programmable buttons but the number of buttons on a remote is limited.

By using gesture interaction, a particular feature can be coupled directly to a specific gesture, improving the ease of access to certain features. The Wii Remote is ideal for this purpose since it can be used to perform gestures and it has buttons which allow quick access to the most commonly used functionality such as changing the volume.

The use of a large set of simple gestures for accessing all the features of a media center application has several disadvantages. Firstly, a large set of gestures is difficult to remember, making it more difficult for the users to deal with the application. A similar problem occurs if button combinations are used on a normal remote to access features of the media center application. Secondly, the probability that different gestures are similar increases for larger gesture sets. If gestures are too similar, the probability that these gestures are incorrectly recognised increases as well, resulting in bad overall recognition rates.

This issue can be solved with multi-modal or composite gestures. The set of simple gestures can be greatly reduced so that it only contains gestures that differ in a significant way. As a consequence, the recognition rate improves and it is easier for the user to remember these gestures.

Gestures can not only be used to access application functionality but also to make decisions. Everyone is familiar with discussions about which TV program or TV channel to watch or whether

[1]http://www.microsoft.com/windows/windows-media-center/get-started/default.aspx

[2]http://xbmc.org/

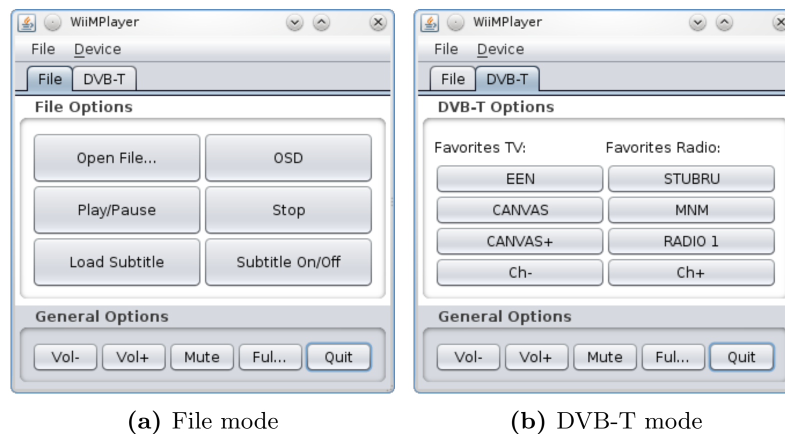
[3]http://www.boxee.tv/

[4]http://www.mythtv.org/

or not to skip a part of a movie. Most of the time, it is the person who holds the remote that has the final word. Even the issue of who holds the remote can lead to some discussion. In order to solve these problems, a voting system could be used. Everyone gets a gesture device (e.g. Wii Remote) and when a decision has to be made everyone that agrees with the decision performs a gesture and the majority decides about the outcome of the discussion.

Implementation

A proof-of-concept application was implemented based on MPlayer[5], an open source media player released under the GPLv2 license. MPlayer is a command line application and therefore it has no GUI. However, it does offer a slave mode[6] which allows an application to start MPlayer in a slave process and send commands to it to control the MPlayer. We implemented a GUI that controls a slave MPlayer process. The GUI is based on the iGesture Workbench to enable fast development and is shown in Figure 5.1.



(a) File mode **(b)** DVB-T mode

Figure 5.1: WiiMPlayer

MPlayer can be used to play multimedia files and to watch TV or listen to the radio via DVB-T. Figure 5.1a shows the File tab which is used to play multimedia files. This tab contains buttons to open a multimedia file, to play, pause and stop playing a file, to load and display subtitles and to show the on-screen display (OSD).

The DVB-T tab shown in Figure 5.1b contains buttons to switch channels and shortcuts to the user's favourite TV and radio channels. Buttons to control general options such as the volume are located at the bottom of the window. Video images are displayed in a separate window that can be maximised to full screen.

The device manager mentioned in Section 4.1 is used to connect the Wii Remote(s) with the application. Each connected Wii Remote sends its gestures to the `Recogniser` which has been configured with the simple gesture set and the Rubine 3D algorithm. The `Recogniser` sends the recognition results to the `MultimodalGestureManager`. The manager sends the gestures forming part of a composite gesture to the `MultimodalGestureRecogniser`, while the other gestures are directly sent to the registered `GestureHandler` which is the main controller. The `MultimodalGestureRecogniser` also hands its recognition results over to the main controller. Based on the recognised gesture, the main

[5]http://www.mplayerhq.hu/design7/news.html
[6]http://www.mplayerhq.hu/DOCS/tech/slave.txt

controller executes the corresponding command. In this proof-of-concept prototype, the mapping between the gesture and the corresponding command is hard-coded and cannot be changed at runtime. In Table 5.1 an example mapping between gestures and commands is shown.

Table 5.1: Mapping between gestures and commands

Gesture	Command
"circle" + "upright"	Open File
"circle" + "S"	Load Subtitle
"square"	Stop
"diagonal"	Full Screen
"square"	CANVAS

The "square" gesture is used two times in Table 5.1 and is given different semantics (i.e. commands) depending on the context. This context is either playing a multimedia file or playing a DVB-T channel. This is also a way to reduce the gesture set and to minimise the load on the user's memory.

To make a decision, a user presses the "A" button on their Wii Remote. The application then shows a window with, for example, the text "Change channel? If yes, please perform the 'triangle' gesture. If no, do nothing.". After 30 seconds, the application shows the results. If a majority is reached, the user can then change the channel.

5.2 Presentation Tool

PaperPoint[7] enables the use of pen and paper-based interaction with Microsoft PowerPoint. The user prints their PowerPoint slides on special paper which is full of patterns almost invisible to the human eye. The Anoto digital pen detects the pattern and sends the coordinates to the PaperPoint application. Everything the user writes on the paper version of their slides is immediately annotated on the digital version as well. Multiple users can annotate the slides at the same time.

PaperPoint offers the possibility to use proximity based constraints. For example, the presenter can draw a "circle" and a "triangle" next to each other. Both figures can be registered as gestures and drawing this sequence close to each other could for example launch an embedded video. Other devices like the Wii Remote could also be used to control video and audio during a presentation, a task that is not easy to do in the current version of PowerPoint. By drawing two chevron-rights, the user could fast forward in the multimedia file. The playback of the file could be paused by drawing two up/down lines.

Using voice commands during a presentation is not straightforward, since it is not clear when a particular word or a combination of words do have to be interpreted as a voice command or as part of the presentation? By performing a gesture in combination with a concurrent or sequential voice command, the correct interpretation can be ensured. Note that the pressing of a button could also be interpreted as a gesture.

Often the presenter has a few slides they find very important or a few extra slides they use as a backup to answer questions. Now the presenter has to enter the number of the slide via the keyboard to jump to that slide. The presenter could map a gesture to each of these slides providing quick access

[7]http://www.globis.ethz.ch/research/paper/applications/paperpoint

to them. Another gesture can be defined as a "go back" function offering the possibility to return to the previously shown slide.

A voting system can also be used during a presentation. For example, if students have already learned about XML in one course and in another course the teacher wants to explain something that is based on XML, the teacher can ask their students whether or not to skip the XML basics. The students then pass their vote by performing a gesture. Of course, this system can also be used for questions with more than one answer. It is possible that all students are provided with a similar device (e.g. a Wii Remote) but the devices do not have to be of the same kind. Nowadays, most students have a mobile phone with a Bluetooth connection and an accelerometer or a touch screen and they could use those devices as well.

5.3 Geco

A last kind of applications that could benefit from multi-modal gesture interaction are application launchers such as Launchy[8], QuickSilver[9], GNOME Do[10] and KRunner[11]. An application launcher offers quick access to applications and locations. To use an application, the user enters a shortcut combination to access the launcher and types in the name of the application or location they want to access and the launcher opens the requested application or location.

Instead of entering a shortcut and a name, why not use gestures? The Gesture Controller (Geco) is an existing command and application launcher that maps gestures to applications or commands as shown in Figure 5.2. Geco is based on the iGesture framework.

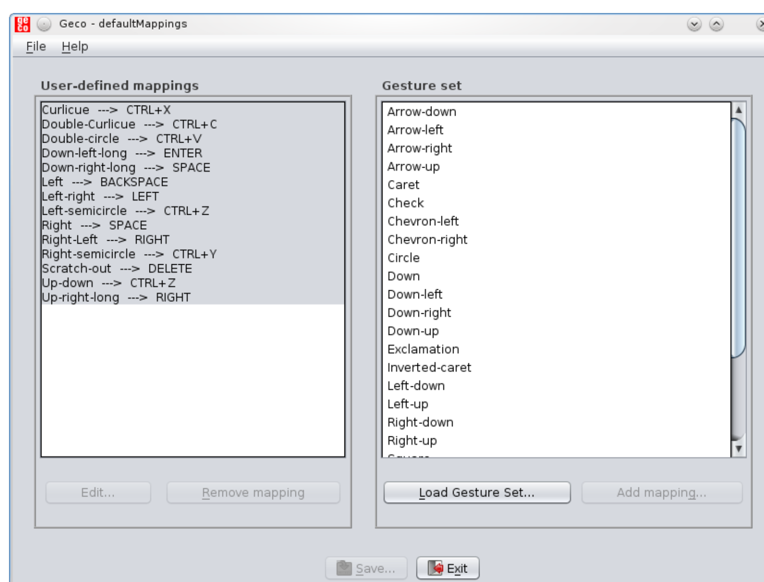


Figure 5.2: Gesture controller (geco)

[8]http://launchy.net/
[9]http://docs.blacktree.com/quicksilver/what_is_quicksilver
[10]http://do.davebsd.com/
[11]http://userbase.kde.org/KRunner

Chapter 6

Summary and Future Work

The goal of this thesis was to extend the iGesture framework with multi-modal gesture interaction functionality. To accomplish this goal, we first integrated support for new gesture devices such as the Wii Remote and TUIO devices (e.g. touch tables) into the framework. Tests show a good overall recognition rate for the Rubine 3D algorithm; however more elaborate testing and a comparison with other algorithms will be required. A device manager component was developed to manage the different devices and their users. Multiple interfaces were defined for the implementation of a device manager and device discovery services.

We have furthermore defined a set of constraints to declaratively combine gestures in multi-modal composite gestures:

- concurrent: concurrent gestures
- sequence: a sequence of gestures
- proximity and concurrent: concurrent gestures that are performed close to each other
- proximity and sequence: a sequence of gestures that are performed close to each other
- interval : multiple gestures that are performed within a given time interval
- cardinality : within a time interval, a particular gesture must be performed between a minimum and a maximum number of times. Each gesture must be performed by a different device or user.

Time, distance, users and devices are the parameters that are necessary to define these constraints. Note that new constraints can be easily added in the future. An XML Schema was defined to specify an XML format for the persistent storage of gesture sets, classes, descriptors and constraints. The same XML Schema can also be used to convert between third-party formats and the format used by iGesture.

To recognise the multi-modal gestures, a multi-modal recogniser has been developed. This recogniser uses a two-phase recognition algorithm. In the first phase, a fuzzy pattern matching algorithm is applied to a gesture input queue to find potential matches. The constraint conditions are then validated in a second phase to determine if a real match has been found. A multi-modal gesture manager is placed between the multi-modal recogniser and other recognisers to make sure that only gestures that potentially form part of a composite gesture are put into the input queue of the multi-modal recogniser. Any other gesture is immediately forwarded to the registered gesture handlers.

Finally, the iGesture Workbench has been modified to support the use of different devices by integrating a device manager. It is now possible to select an input device without any recompilation.

The Workbench was extended with a graphical user interface to define the multi-modal composite gestures in a declarative manner and to test the recognition performance of these gestures. We have also mentioned multiple applications where multi-modal gesture interaction gives an added value such as multimedia players, presentation tools and application launchers.

During my research, I have investigated a lot of related frameworks and technologies (e.g. multi-modal fusion engines). In the practical part of this project, I also learned how to deal with large projects and frameworks by using dependency management tools such as Maven[1].

Future Work

There is still a lot work that can be done to extend and improve the iGesture framework. Throughout this thesis, some of these potential extensions have already been mentioned. New gesture recognition algorithms could be added based on the algorithms of the WEKA tool. A rule-based or a multi-modal fusion engine could be investigated as an alternative way of recognising composite gestures and modalities such as voice recognition might be added as well. The recognition rate of the Wii Remote might be further improved by making use of the orientation data provided by the new Wii MotionPlus extension.

Support for web services could be added by supporting the EMMA markup language. Last but not least, a JGraph-based GUI to define and edit composite gestures and the development of a component to define composite gestures by actively performing them may also form part of future efforts

[1] http://maven.apache.org/

Appendix A

UML Diagrams

In this appendix, the UML class diagrams of different parts of the framework can be consulted.

- Device Manager on page 73
- TUIO on page 78
- Multi-modal gestures and components on page 83

A.1 Device Manager

Figure A.1: Package diagram for the device manager related classes

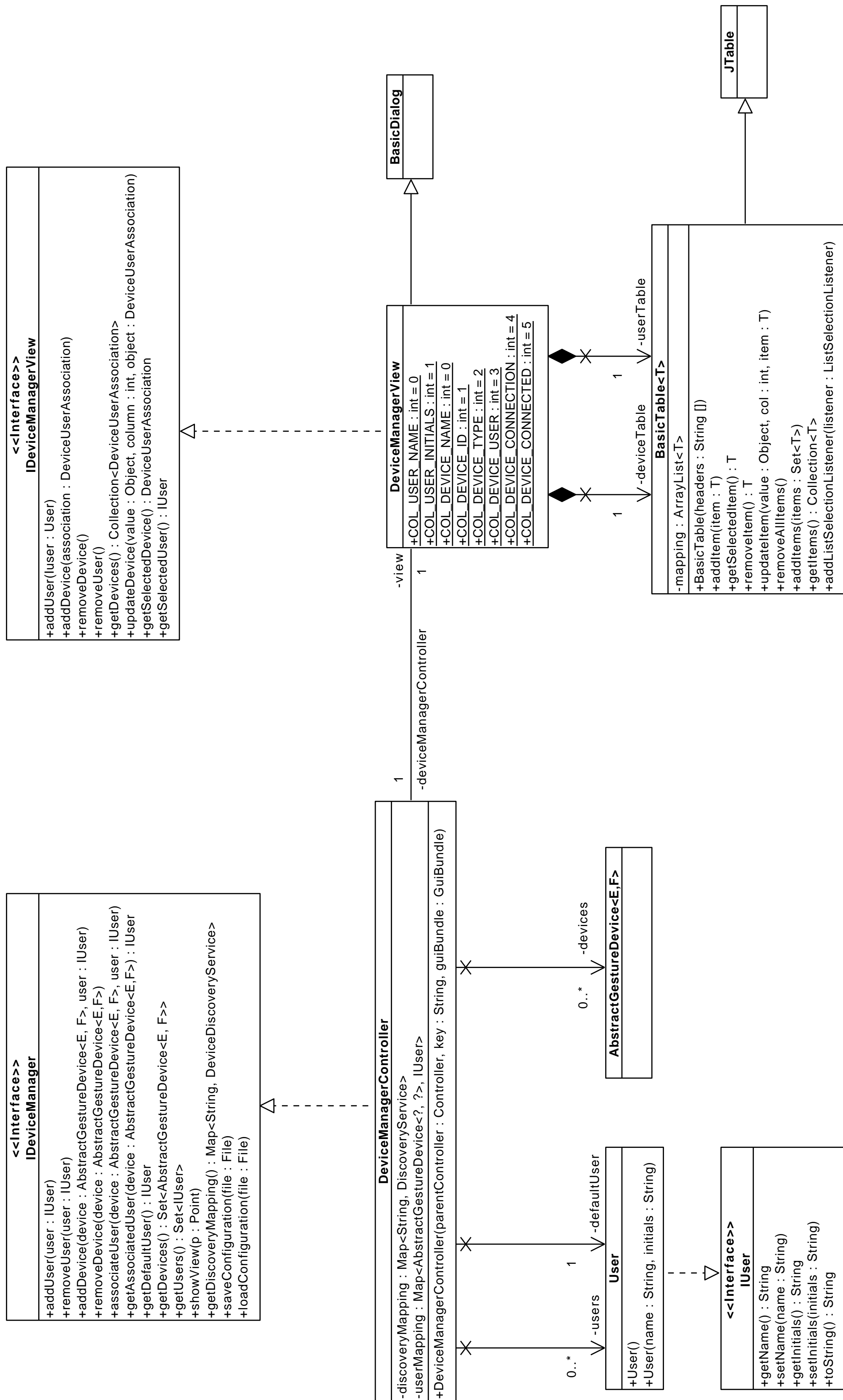


Figure A.2: Class diagram for the device manager

Figure A.3: Class diagram for the device and user related classes

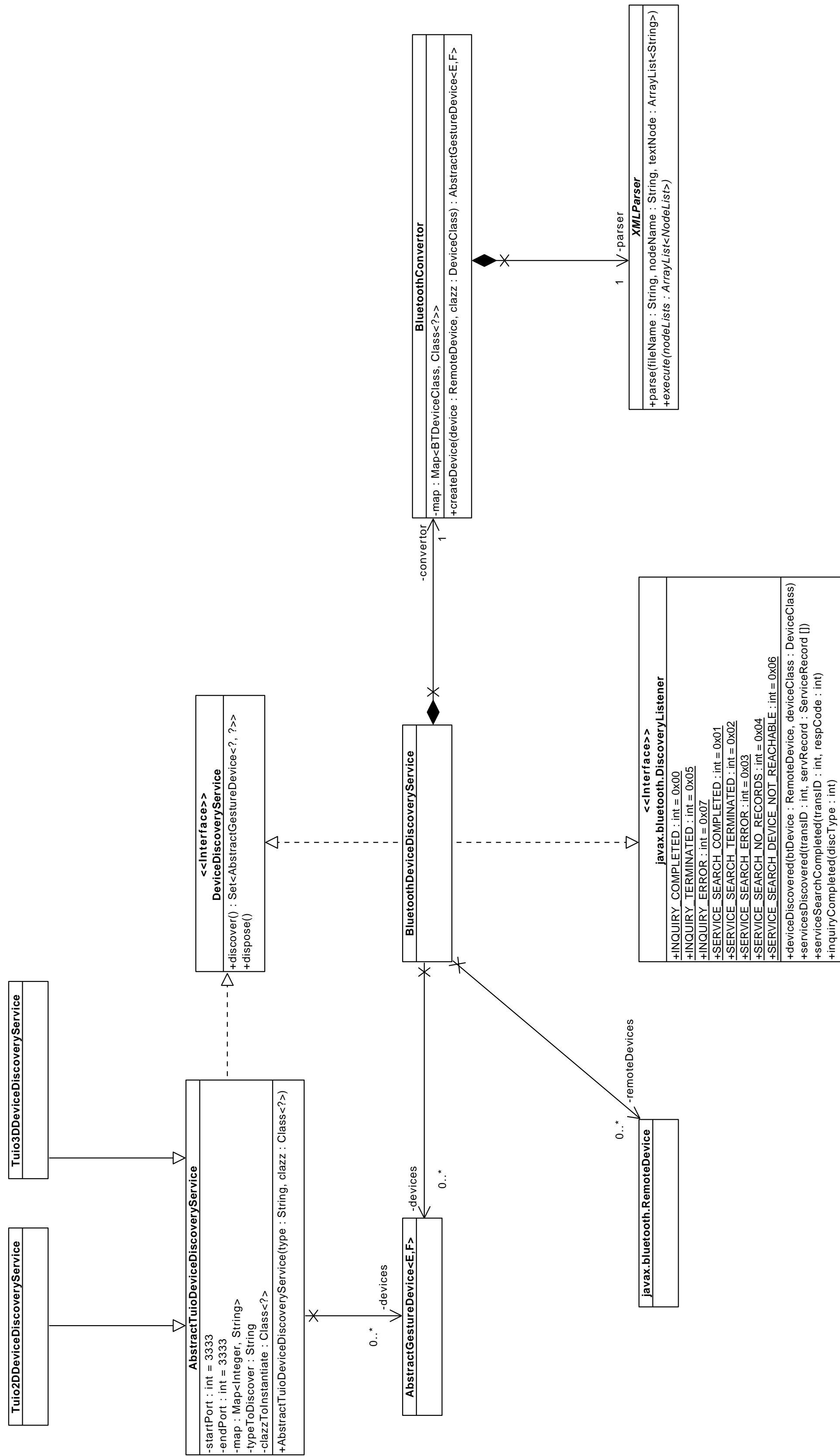


Figure A.4: Class diagram for the device discovery service related classes

Figure A.5: Class diagram for the Add Device wizard and the device manager actions

A.2 TUIO

Figure A.6: Package diagram for the TUIO related classes

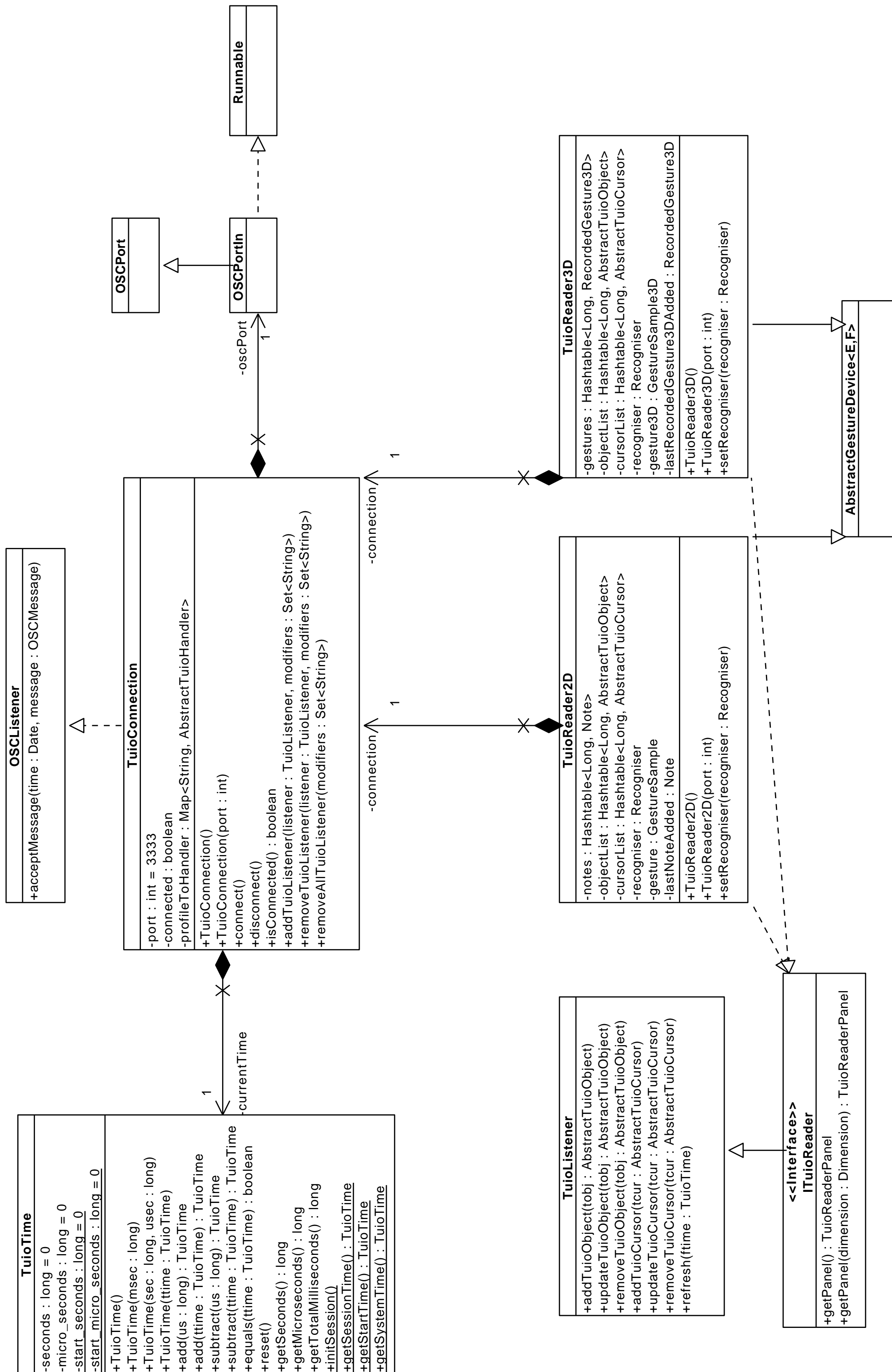


Figure A.7: Class diagram for the TuioConnection and TuioReader related classes

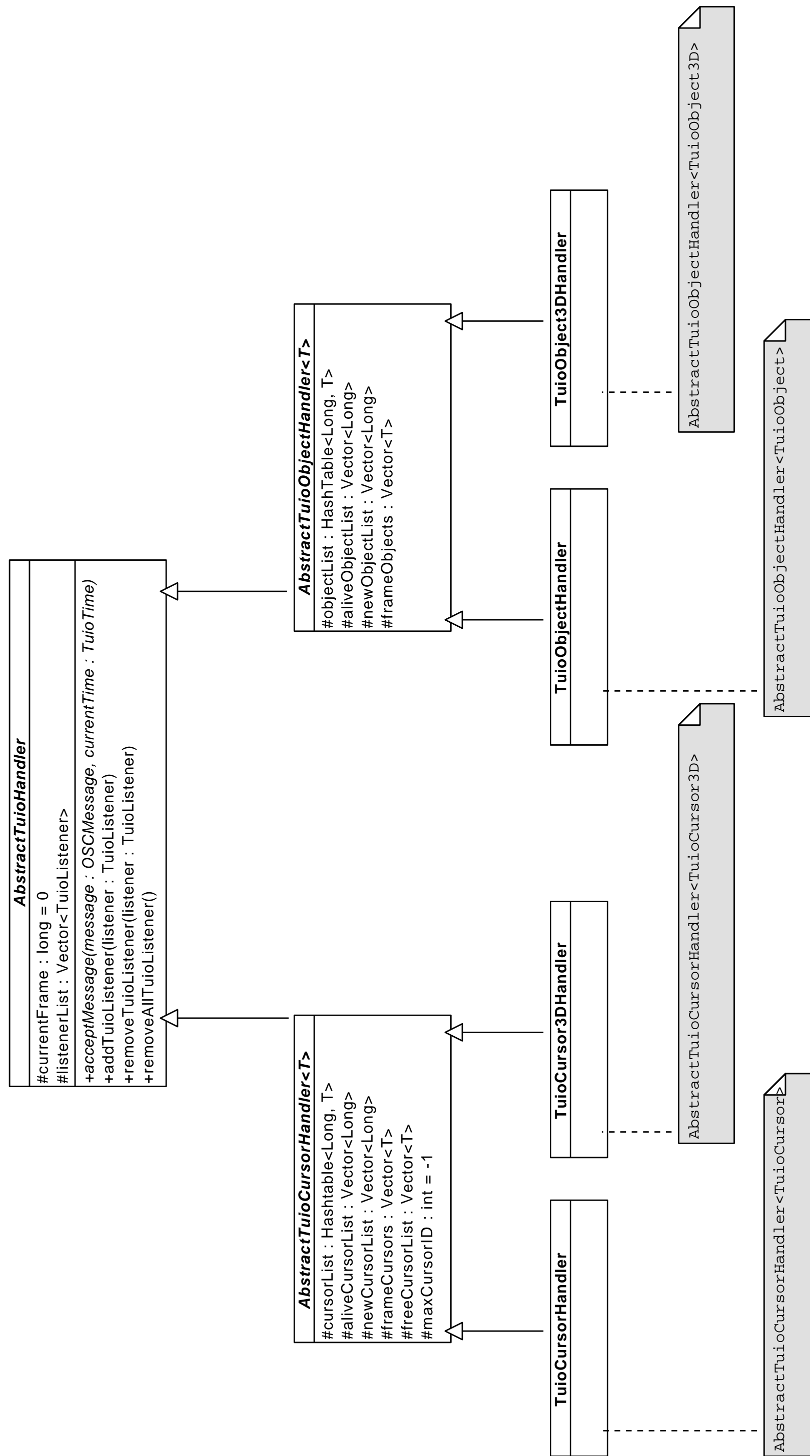


Figure A.8: Class diagram for the TuioHandler classes

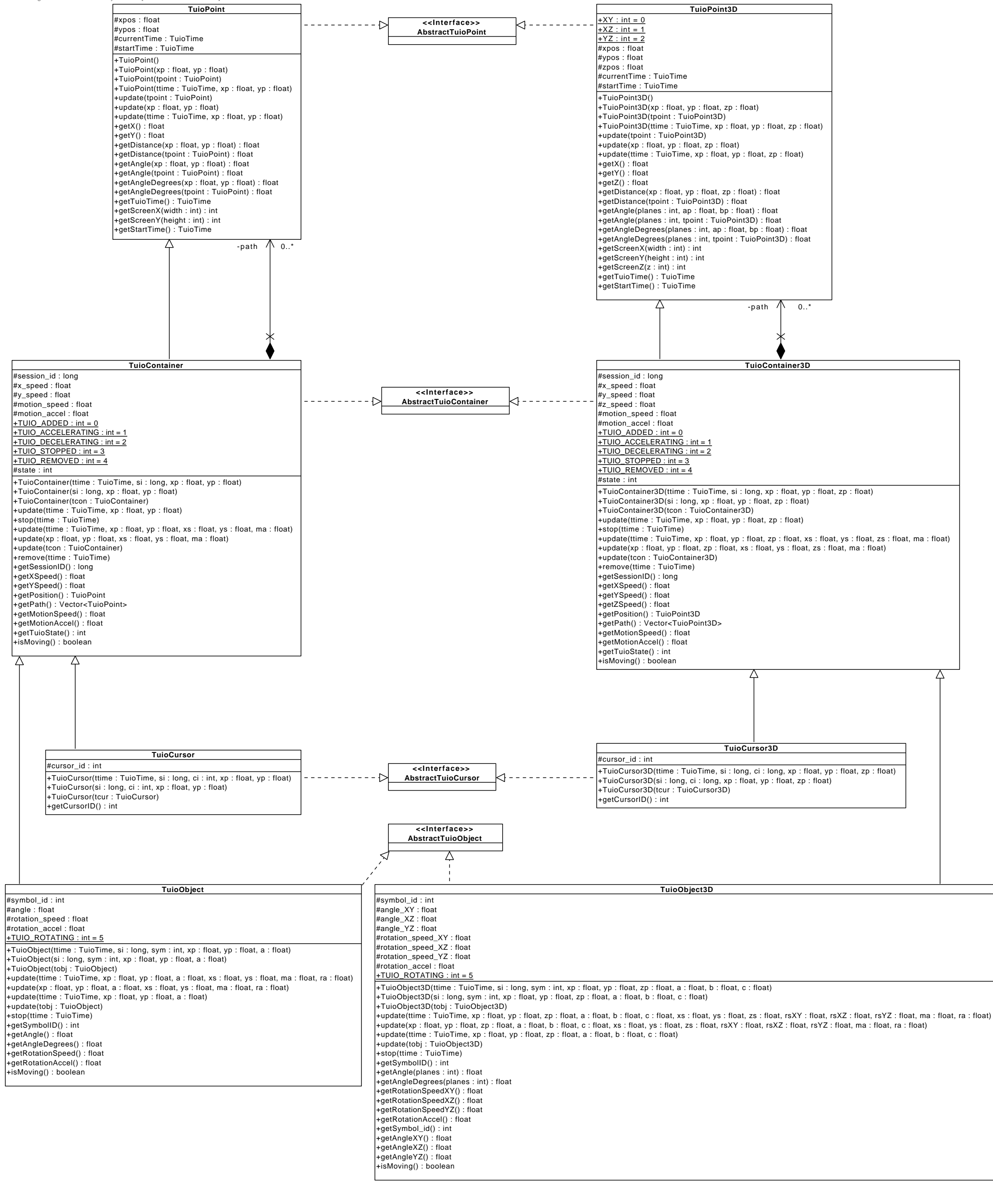


Figure A.9: Class diagram for the TuioCursor and TuioObject related classes

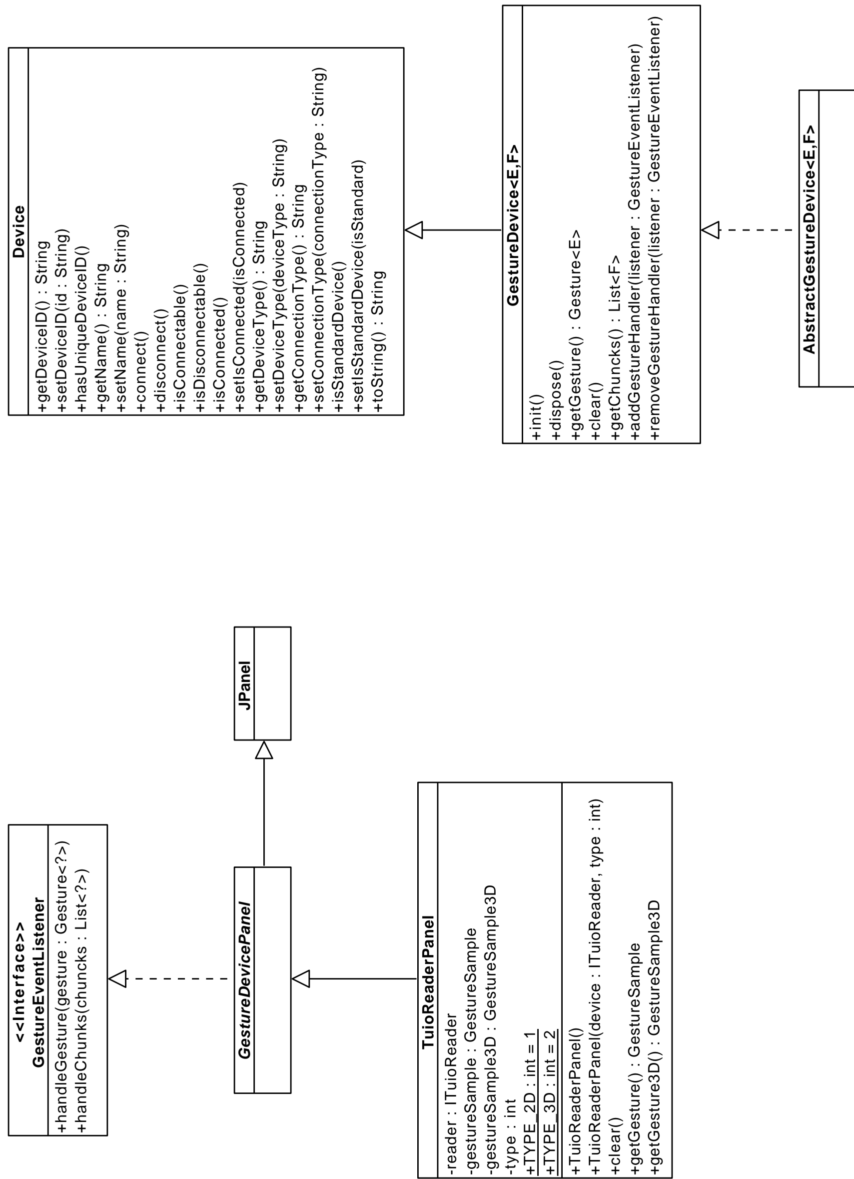


Figure A.10: Class diagram for the TuioReaderPanel and Device interface related classes

A.3 Multi-modal Gestures

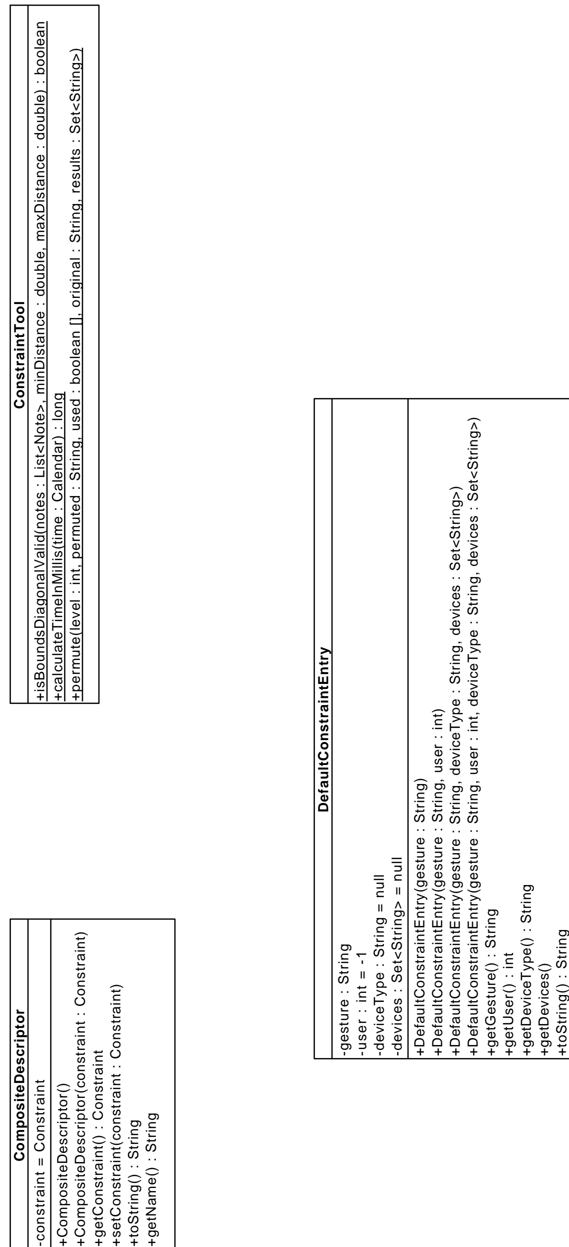


Figure A.11: Class diagram for the other multi-modal gesture related Classes

Figure A.12: Class diagram for the constraints

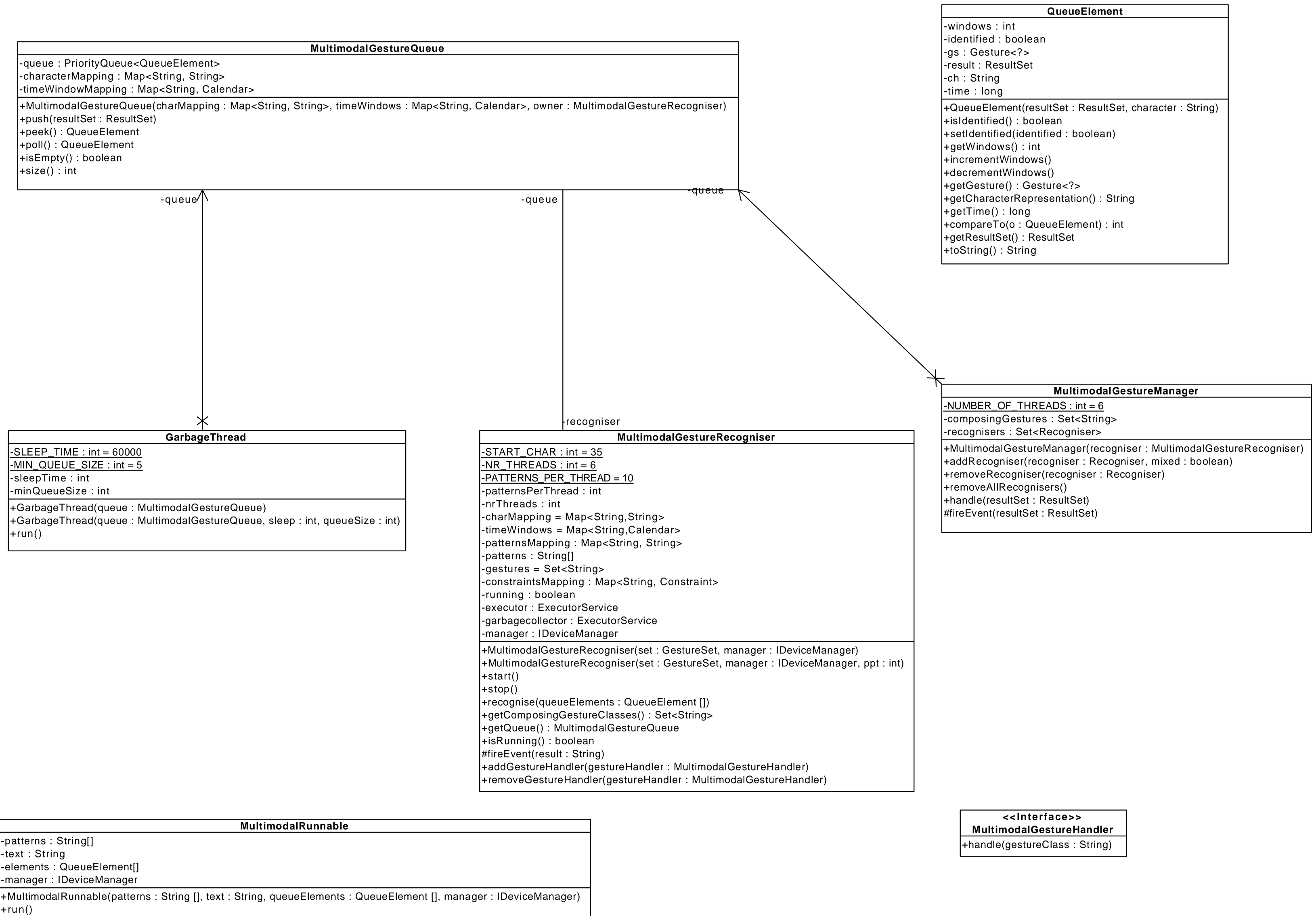


Figure A.13: Class diagram for the multi-modal components

Appendix B

XML Schema

The XML Schema is divided into three parts. The first part (`iGestureSet.xsd`) describes the gesture sets and gesture classes. `Descriptor.xsd` describes the different descriptors such as the sample, text or composite descriptor. Finally, `constraint.xsd` describes the different constraints that have been defined to composite gestures.

B.1 iGestureSet.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://igesture.org/xml/1/iGestureSet"
 elementFormDefault="qualified"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:igs="http://igesture.org/xml/1/iGestureSet">
  <include schemaLocation="http://igesture.org/xml/1/constraint.xsd"/>
  <include schemaLocation="http://igesture.org/xml/1/descriptor.xsd"/>

  <element name="sets" type="igs:GestureSetType">
    <annotation>
      <documentation>Representation of a gesture or test set.</documentation>
    </annotation>

    <!-- UNIQUENESS of IDs -->
    <unique name="uniqueId">
      <selector xpath="igs:class | igs:set | igs:class/igs:descriptor |
           igs:class/igs:descriptor/igs:constraint | igs:class/igs:descriptor/
           igs:sample"/>
      <field xpath="@id"/>
    </unique>

    <!-- REFERENCE KEYS: EXIST + UNIQUE -->
    <key name="OnToClassOrConstraint">
      <selector xpath="igs:class | igs:class/igs:descriptor/igs:constraint"/>
      <field xpath="@id"/>
    </key>
    <keyref name="OnToClassOrConstraintRef" refer="igs:OnToClassOrConstraint">
      <selector xpath="igs:class/igs:descriptor/igs:constraint/igs:on"/>
      <field xpath="@idref"/>
    </keyref>

    <key name="SetClassToClass">
      <selector xpath="igs:class"/>
      <field xpath="@id"/>
    </key>
    <keyref name="SetClassToClassRef" refer="igs:SetClassToClass">
      <selector xpath="igs:set/igs:class"/>
      <field xpath="@idref"/>
    </keyref>
  </element>

  <complexType name="GestureSetType">
    <sequence>
      <element name="class" type="igs:ClassType" maxOccurs="unbounded" />
      <element name="set" type="igs:SetType" maxOccurs="unbounded" />
    </sequence>
    <attribute name="type" use="optional" default="gesture">
      <simpleType>
        <restriction base="string">
          <enumeration value="gesture"/>
          <enumeration value="test"/>
        </restriction>
      </simpleType>
    </attribute>
  </complexType>

  <complexType name="ClassType">
    <sequence>
```

```
      <element name="descriptor" type="igs:DescriptorType"/>
    </sequence>
    <attributeGroup ref="igs:NameIdGroup"/>
  </complexType>

  <complexType name="SetType">
    <sequence>
      <element name="class" maxOccurs="unbounded">
        <complexType>
          <attribute name="idref" type="igs:IDType" use="required"/>
        </complexType>
      </element>
    </sequence>
    <attributeGroup ref="igs:NameIdGroup"/>
  </complexType>

  <attributeGroup name="NameIdGroup">
    <attribute name="name" type="string" use="required"/>
    <attribute name="id" type="igs:IDType" use="required"/>
  </attributeGroup>

  <attributeGroup name="TypeIdGroup">
    <attribute name="id" type="igs:IDType" use="required"/>
    <attribute name="type" type="string" use="required"/>
  </attributeGroup>

  <simpleType name="IDType">
    <restriction base="string">
      <pattern value="\w{8}-\w{4}-\w{4}-\w{4}-\w{12}"/>
    </restriction>
  </simpleType>
</schema>
```

B.2 descriptor.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://igesture.org/xml/1/iGestureSet"
 elementFormDefault="qualified"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:igs="http://igesture.org/xml/1/iGestureSet" >

  <complexType name="DescriptorType">
    <annotation>
      <documentation>Generic Descriptor.</documentation>
    </annotation>
    <attribute name="id" type="igs:IDType" use="required" />
  </complexType>

  <complexType name="PointType">
    <annotation>
      <documentation>Generic Point.</documentation>
    </annotation>
  </complexType>

  <complexType name="Point2DType">
    <annotation>
      <documentation>Representation of a 2D point.</documentation>
    </annotation>
    <complexContent>
      <extension base="igs:PointType">
        <sequence>
          <element name="timestamp" type="dateTime">
            <annotation>
              <documentation>
                Timestamp conform to IS0-8601 format.
              </documentation>
            </annotation>
          </element>
          <element name="x" type="double"/>
          <element name="y" type="double"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="Point3DType">
    <annotation>
      <documentation>Representation of a 3D point.</documentation>
    </annotation>
    <complexContent>
      <extension base="igs:Point2DType">
        <sequence><element name="z" type="double"/></sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="TraceType">
    <sequence>
      <element name="point" type="igs:PointType" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="Note3DType">
```

```
  <annotation>
    <documentation>Representation of a 3D gesture.</documentation>
  </annotation>
  <sequence>
    <element name="point3D" type="igs:Point3DType" maxOccurs="unbounded"/>
    <element name="acceleration" type="igs:AccelerationType" minOccurs="0"/>
  </sequence>
</complexType>

<complexType name="AccelerationType">
  <sequence>
    <element name="sample" type="igs:AccelerationSampleType" maxOccurs="
         unbounded"/>
  </sequence>
</complexType>

<complexType name="AccelerationSampleType">
  <sequence>
    <element name="timestamp" type="dateTime"/>
    <element name="xAcc" type="double"/>
    <element name="yAcc" type="double"/>
    <element name="zAcc" type="double"/>
  </sequence>
</complexType>

<complexType name="SampleDescriptorType">
  <complexContent>
    <extension base="igs:DescriptorType">
      <sequence>
        <element name="sample" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="note">
                <complexType>
                  <sequence>
                    <element name="trace" type="igs:TraceType" maxOccurs="
                         unbounded"/>
                  </sequence>
                </complexType>
              </element>
            </sequence>
            <attributeGroup ref="igs:NameIdGroup"/>
          </complexType>
        </element>
      </sequence>
      <attribute name="type" type="string" use="required"
        fixed="org.ximtec.igesture.core.SampleDescriptor">
      </attribute>
    </extension>
  </complexContent>
</complexType>

<complexType name="SampleDescriptor3DType">
  <complexContent>
    <extension base="igs:DescriptorType">
      <sequence>
        <element name="sample" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="note3D" type="igs:Note3DType"/>
```

```
          </sequence>
          <attributeGroup ref="igs:NameIdGroup"/>
        </complexType>
      </element>
    </sequence>
    <attribute name="type" type="string" use="required"
      fixed="org.ximtec.igesture.core.SampleDescriptor3D">
    </attribute>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="CompositeDescriptorType">
  <complexContent>
   <extension base="igs:DescriptorType">
    <sequence>
      <element name="constraint" type="igs:ConstraintType" />
    </sequence>
    <attribute name="type" type="string" use="required"
      fixed="org.ximtec.igesture.core.CompositeDescriptor">
    </attribute>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="TextDescriptorType">
  <complexContent>
   <extension base="igs:DescriptorType">
    <sequence>
      <element name="text">
        <simpleType>
          <restriction base="string">
            <pattern value="[N,S,E,W]+"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
    <attribute name="type" type="string" use="required"
      fixed="org.ximtec.igesture.core.TextDescriptor"/>
   </extension>
  </complexContent>
 </complexType>
</schema>
```

B.3 constraint.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://igesture.org/xml/1/iGestureSet"
 elementFormDefault="qualified"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:igs="http://igesture.org/xml/1/iGestureSet">

  <complexType name="ConstraintType">
    <annotation>
      <documentation>Generic Constraint</documentation>
    </annotation>
    <sequence>
      <element name="gesture" maxOccurs="unbounded">
        <complexType>
          <annotation>
            <documentation>
              A gesture can be performed by a particular user.
              The user is referenced by a number because the user is not
              known at the time of definition and to allow flexibility.
              It is also possible to define the type of device used to perform
                  the gesture.
            </documentation>
          </annotation>
          <attribute name="id" type="igs:IDType" use="required"/>
          <attribute name="idref" type="igs:IDType" use="required">
            <annotation>
              <documentation>A constraint can refer to a gesture class</
                  documentation>
            </annotation>
          </attribute>
          <attribute name="user" type="int" use="optional" default="0" />
          <attribute name="device" type="string" use="optional" />
        </complexType>
      </element>
      <element name="devices" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="devicename" type="string" maxOccurs="unbounded"/>
          <element name="idref">
            <simpleType><list itemType="igs:IDType" /></simpleType>
          </element>
        </sequence>
        </complexType>
      </element>
    </sequence>
    <attribute name="id" type="igs:IDType" use="required" />
  </complexType>

    <complexType name="AbstractCardinalityConstraintType">
      <annotation>
        <documentation>
          Constraint where the gesture and the devices element can only appear
              once.
          The user attribute can not be used. Used for cardinality constraints.
        </documentation>
      </annotation>
      <complexContent>
        <restriction base="igs:ConstraintType">
          <sequence>
```

```
          <element name="gesture" minOccurs="1" maxOccurs="1">
            <complexType>
              <attribute name="id" type="igs:IDType" use="required"/>
              <attribute name="idref" type="igs:IDType" use="required">
                <annotation>
                  <documentation>
                    A constraint can refer to a gesture class
                  </documentation>
                </annotation>
              </attribute>
              <attribute name="device" type="string" use="optional" />
            </complexType>
          </element>
          <element name="devices" minOccurs="0" maxOccurs="1">
            <complexType>
              <sequence>
                <element name="devicename" type="string" maxOccurs="unbounded"
                    />
                <element name="idref">
                  <simpleType><list itemType="igs:IDType" /></simpleType>
                </element>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </restriction>
    </complexContent>
  </complexType>

  <complexType name="AbstractProximityConstraintType">
    <annotation>
      <documentation>
        Constraint where the device attribute is required, used for proximity
            constraints
      </documentation>
    </annotation>
    <complexContent>
      <restriction base="igs:ConstraintType">
        <sequence>
          <element name="gesture" minOccurs="1" maxOccurs="unbounded">
            <complexType>
              <attribute name="id" type="igs:IDType" use="required"/>
              <attribute name="idref" type="igs:IDType" use="required">
                <annotation>
                  <documentation>
                    A constraint can refer to a gesture class
                  </documentation>
                </annotation>
              </attribute>
              <attribute name="user" type="int" use="optional" default="0" />
              <attribute name="device" type="string" use="required" />
            </complexType>
          </element>
          <element name="devices" minOccurs="0" maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element name="devicename" type="string" maxOccurs="unbounded"
                    />
                <element name="idref">
                  <simpleType><list itemType="igs:IDType" /></simpleType>
```

```
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </restriction>
</complexContent>
</complexType>

<!-- derived constraint types -->
<complexType name="ProximitySequenceConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
      each other proximity and in sequence.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:AbstractProximityConstraintType">
      <sequence>
        <element name="param" type="igs:ProximitySequenceParamType"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.ProximitySequenceConstraint"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="ProximityConcurrentConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
      each other proximity and concurrent.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:AbstractProximityConstraintType">
      <sequence>
        <element name="param" type="igs:ProximityConcurrentParamType"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.ProximityConcurrencyConstraint"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="ConcurrentConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed
          concurrent.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:ConstraintType">
      <sequence>
        <element name="param" type="igs:ConcurrentParamType"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.ConcurrencyConstraint"/>
```

```
      </extension>
    </complexContent>
  </complexType>

  <complexType name="IntervalConstraintType">
    <annotation>
      <documentation>
        Representation of a constraint where the gestures are performed within
             a certain time interval
      </documentation>
    </annotation>
    <complexContent>
      <extension base="igs:ConstraintType">
        <sequence>
          <element name="param" type="igs:IntervalParamType"/>
        </sequence>
        <attribute name="type" type="string" use="required" fixed="org.ximtec.
            igesture.core.composite.IntervalConstraint"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="CardinalityDeviceConstraintType">
    <annotation>
      <documentation>
        Representation of a constraint where the gestures are performed
        within a certain time interval and there is a lower and an upper
        limit of how many times the gesture has to be performed.
        Each device has one vote.
      </documentation>
    </annotation>
    <complexContent>
      <extension base="igs:AbstractCardinalityConstraintType">
        <sequence>
          <element name="param" type="igs:CardinalityParamType"/>
        </sequence>
        <attribute name="type" type="string" use="required" fixed="org.ximtec.
            igesture.core.composite.CardinalityDeviceConstraint"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="CardinalityUserConstraintType">
    <annotation>
      <documentation>
        Representation of a constraint where the gestures are performed
        within a certain time interval and there is a lower and an upper
        limit of how many times the gesture has to be performed.
        Each user has one vote.
      </documentation>
    </annotation>
    <complexContent>
      <extension base="igs:AbstractCardinalityConstraintType">
        <sequence>
          <element name="param" type="igs:CardinalityParamType"/>
        </sequence>
        <attribute name="type" type="string" use="required" fixed="org.ximtec.
            igesture.core.composite.CardinalityUserConstraint"/>
      </extension>
    </complexContent>
```

```
</complexType>

<complexType name="SequenceConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
          sequence.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:ConstraintType">
      <sequence>
        <element name="param" type="igs:SequenceParamType"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.SequenceConstraint"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="XProximitySequenceConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
      each other proximity and in sequence. Each gesture can have its own
          parameters.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:AbstractProximityConstraintType">
      <sequence>
        <element name="param" type="igs:XProximitySequenceParamType"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.XProximitySequenceConstraint"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="XProximityConcurrentConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
      each other proximity and concurrent. Each gesture can have its own
          parameters.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:AbstractProximityConstraintType">
      <sequence>
        <element name="param" type="igs:XProximityConcurrentParamType"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.XProximityConcurrencyConstraint"/>
    </extension>
  </complexContent>
</complexType>
```

```
<complexType name="XSequenceConstraintType">
  <annotation>
    <documentation>
      Representation of a constraint where the gestures are performed in
      sequence. Each gesture can have its own parameters.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="igs:ConstraintType">
      <sequence>
        <element name="param" type="igs:XSequenceParamType" maxOccurs="
            unbounded"/>
      </sequence>
      <attribute name="type" type="string" use="required" fixed="org.ximtec.
          igesture.core.composite.XSequenceConstraint"/>
    </extension>
  </complexContent>
</complexType>

<!-- parameter types -->
<complexType name="ProximitySequenceParamType">
  <complexContent>
    <extension base="igs:SequenceParamType">
      <sequence>
        <element name="minDistance" type="double"/>
        <element name="maxDistance" type="double"/>
        <element name="distanceUnit" type="igs:DistanceUnitType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="SequenceParamType">
  <sequence>
    <element name="minTime" type="time"/>
    <element name="maxTime" type="time"/>
  </sequence>
</complexType>

<complexType name="IntervalParamType">
  <sequence>
    <element name="time" type="time"/>
  </sequence>
</complexType>

<complexType name="CardinalityParamType">
  <complexContent>
    <extension  base="igs:IntervalParamType">
      <sequence>
        <element name="min" type="int" />
        <element name="max" type="int" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="ProximityConcurrentParamType">
  <sequence>
    <element name="minDistance" type="double"/>
    <element name="maxDistance" type="double"/>
```

```
      <element name="distanceUnit" type="igs:DistanceUnitType"/>
    </sequence>
  </complexType>

  <complexType name="XProximitySequenceParamType">
    <complexContent>
      <extension base="igs:XSequenceParamType">
        <sequence>
          <element name="idref" type="igs:ParamUnionType"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="XSequenceParamType">
    <complexContent>
      <extension base="igs:SequenceParamType">
        <sequence>
          <element name="idref" type="igs:ParamUnionType"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="XProximityConcurrentParamType">
    <complexContent>
      <extension base="igs:ProximityConcurrentParamType">
        <sequence>
          <element name="idref" type="igs:ParamUnionType"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <simpleType name="ParamUnionType">
    <union memberTypes="igs:ParamDefaultType igs:ParamListType"/>
  </simpleType>

  <simpleType name="ParamDefaultType">
    <restriction base="string">
      <enumeration value="default"/>
    </restriction>
  </simpleType>

  <simpleType name="ParamListType">
    <list itemType="igs:IDType"/>
  </simpleType>

  <simpleType name="DistanceUnitType">
    <restriction base="string">
      <enumeration value="km"/>
      <enumeration value="m"/>
      <enumeration value="cm"/>
      <enumeration value="mm"/>
    </restriction>
  </simpleType>
</schema>
```

Bibliography

[1] Ricardo Baeza-Yates and Gaston H. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[2] Paolo Baggia, Daniel C. Burnett, Jerry Carter, Deborah A. Dahl, Gerry McCobb, and Dave Raggett. EMMA: Extensible MultiModal Annotation Markup Language. W3C Recommendation, January 2009.

[3] Srinivas Bangalore and Michael Johnston. Robust Gesture Processing for Multimodal Interaction. In *Proceedings of ICMI 2008, 10th International Conference on Multimodal Interfaces*, pages 225–232, Chania,Greece, October 2008.

[4] Marcello Bastea-Forte, Ron Yeh M., and Scott R. Klemmer. Pointer: Multiple Collocated Display Inputs Suggests New Models for Program Design and Debugging. In *Proceedings of UIST 2007, 20th Annual ACM Symposium on User Interface Software and Technology*, Newport, USA, October 2007.

[5] Benjamin B. Bederson and Jason Stewart Allison Druin. Single Display Groupware. Technical Report UMIACS-TR-99-75, UM Computer Science Department, December 1999.

[6] Ari Y. Benbasat and Joseph A. Paradiso. An Inertial Measurement Framework for Gesture Recognition and Applications. In *Revised Papers from the International Gesture Workshop on Gesture and Sign Languages in Human-Computer Interaction, LNCS 2298*, pages 9–20, 2001.

[7] Richard A. Bolt. "Put-That-There": Voice and Gesture at the Graphics Interface. In *Proceedings of ACM SIGGRAPH 80, 7th International Conference on Computer Graphics and Interactive Techniques*, pages 262–270, Seattle, USA, July 1980.

[8] Jullien Bouchet, Laurence Nigay, and Thierry Ganill. ICARE Software Components for Rapidly Developing Multimodal Interfaces. In *Proceedings of ICMI 2004, 6th International Conference on Multimodal Interfaces*, pages 251–258, State College, USA, October 2004.

[9] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[10] Microsoft Corporation. Experience Microsoft Surface. White Paper, 2008.

[11] Bruno Dumas, Denis Lalanne, and Sharon Oviatt. Multimodal Interfaces: A Survey of Principles, Models and Frameworks. *Human Machine Interaction, LNCS 5440*, pages 3–26, 2009.

[12] Geoffrey Holmes, Andrew Donkin, and Ian H. Witten. WEKA: A Machine Learning Workbench. In *Proceedings of the 2nd Australian and New Zealand Conference on Intelligent Information Systems*, pages 357–361, Brisbane, Australia, November 1994.

[13] Peter Hutterer and Bruce H. Thomas. Groupware Support in the Windowing System. In *Proceedings of AUIC 2007, 8th Eighth Australasian User Interface Conference*, pages 39–46, Ballarat, Australia, January 2007.

[14] Michael Johnston. Building Multimodal Applications with EMMA. In *Proceedings of IMCI 2009, 11th International Conference on Multimodal Interfaces*, pages 47–54, Cambridge, USA, November 2009.

[15] Michael Johnston, Srinivas Bangalore, Gunaranjan Vasireddy, Amanda Stent, Patrick Ehlen, Marilyn Walker, Steve Whittaker, and Preetam Maloor. MATCH: An Architecture for Multimodal Dialogue Systems. In *Proceedings of ACL 2002, 40th Annual Meeting on Association for Computational Linguistics*, pages 376–383, Philadelphia, USA, 2002.

[16] Ricardo Jota, Bruno R. de Araújo, Luís C. Bruno, João M. Pereira, and Joaquim A. Jorge. IMMIView: A Multi-user Solution for Design Review in Real-time. *Journal of Real-Time Image Processing*, November 2009.

[17] Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. TUIO: A Protocol for Table-Top Tangible User Interfaces. In *Proceedings of GW 2005, 6th International Workshop on Gesture in Human-Computer Interaction and Simulation*, Ile de Berder, France, May 2005.

[18] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[19] Anna Koster. WiiCon: Acceleration Based Real-Time Conducting Gesture Recognition for Personal Orchestra. Master's thesis, RWTH Aachen, December 2008.

[20] Robert Neßelrath and Jan Alexandersson. A 3D Gesture Recognition System for Multimodal Dialog Systems. In *Proceedings of IJCAI 2009, 6th IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, pages 46–51, Pasadena, USA, July 2009.

[21] Nokia. Next Generation User Interfaces. White Paper, 2010.

[22] Sharon Oviatt, Phil Cohen, Lizhong Wu, Lisbeth Duncan, Bernhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and David Ferro. Designing the User Interface for Multimodal Speech and Gesture Applications: State-of-the-Art Systems and Research Directions. *Human Computer Interaction*, 15(4):263–322, 2000.

[23] Dean Rubine. Specifying Gestures by Example. In *Proceedings of ACM SIGGRAPH '91, 18th International Conference on Computer Graphics and Interactive Techniques*, pages 329–337, 1991.

[24] Thomas Schlömer, Benjamin Poppinga, Niels Henze, and Susanne Boll. Gesture Recognition with a Wii Controller. In *Proceedings of TEI 2008, 2nd International Conference on Tangible and Embedded Interaction*, pages 11–14, Bonn, Germany, February 2008.

[25] Garth B. D. Shoemaker and Kori M. Inkpen. MIDDesktop: An Application Framework for Single Display Groupware Investigations. Technical Report TR 2000-1, Simon Fraser University, April 2001.

[26] Beat Signer, Ueli Kurmann, and Moira C. Norrie. iGesture: A General Gesture Recognition Framework. In *Proceedings of ICDAR 2007, 9th International Conference on Document Analysis and Recognition*, pages 954–958, Curitiba, Brazil, September 2007.

[27] Beat Signer, Moira C. Norrie, and Ueli Kurmann. iGesture: A Java Framework for the Development and Deployment of Stroke-Based Online Gesture Recognition Algorithms. Technical Report TR561, ETH Zurich, CH-8092 Zurich, Switzerland, September 2007.

[28] Jason Stewart, Benjamin B. Bederson, and Allison Druin. Single Display Groupware: A Model for Co-present Collaboration. In *Proceedings of CHI '99, ACM Conference on Human Factors in Computing Systems*, pages 286–293, Pittsburgh, USA, May 1999.

[29] Edward Tse. The Single Display Groupware Toolkit. Master's thesis, University of Calgary, November 2004.

[30] Edward Tse and Saul Greenberg. Rapidly prototyping single display groupware through the sdgtoolkit. In *Proceedings of AUIC 2004, 5th Eighth Australasian User Interface Conference*, pages 101–110, Dunedin, New Zealand, January 2004.

[31] Arthur Vogels. iGesture Extension for 3D Recognition. Project Report, 2009.

[32] Grant Wallace, Peng Bi, Kai Li, and Otto Anshus. A Multi-Cursor X Window Manager Supporting Control Room Collaboration. Technical Report TR-707-04, Princeton University, Department of Computer Science, July 2004.

[33] Wikipedia. http://en.wikipedia.org/wiki/Computer_supported_cooperative_work, 2010.

[34] Daniel Wilson and Andy Wilson. Gesture Recognition Using the XWand. Technical Report CMU-RI-TR-04-57, Carnegie Mellon University, Pittsburgh, PA, April 2004.

References

The following pictures were found and taken from the mentioned papers, programs or websites.

Introduction

Figure 1.1 on page 1 - Visual Paradigm UML Editor
Figure 1.2 on page 2 - http://www.visual-paradigm.com/support/documents/vpumluserguide/26/31/6787_mousegesture.html

Background

Figure 2.1 on page 5 - http://en.wikipedia.org/wiki/CSCW
Figure 2.2 on page 6 - [25]
Figure 2.3 op page 6 - [29]
Figure 2.4 on page 8 - [4]
Table 2.1 op page 10 - [11]
Figure 2.5 on page 11 - [11]
Figure 2.6 on page 11 - [11]
Figure 2.9 on page 14 - [14]
Figure 2.7 on page 12 - [11]
Figure 2.10 on page 15 - [16]
Figure 2.11 on page 16 - [8]
Figure 2.8 on page 13 - [11]
Figure 2.12 on page 16 - http://www.crunchgear.com/2010/03/11/the-playstation-move-everything-old-is-new-again-if-you-ask-sony/
Figure 2.13 on page 17 - http://tweakers.net/nieuws/61265/microsoft-wil-natal-bewegingsdetectie-ook-geschikt-maken-voor-pc.html
Figure 2.14 on page 17 - http://blogs.msdn.com/surface/archive/2007/11/01/surface-computing-has-

arrived-and-so-has-our-blog.aspx
Figure 2.15 on page 18 - Visual Paradigm UML Editor
Figure 2.16 on page 19 - [21]
Figure 2.17a on page 19 - http://www.apple.com/iphone/how-to/#basics.scrolling
Figure 2.17c on page 19 - http://www.apple.com/iphone/how-to/#basics.zooming-in-or-out
Figure 2.17b on page 19 - http://www.apple.com/iphone/how-to/#basics.scrolling
Figure 2.18 on page 19 - http://www.apple.com/magicmouse/
Figure 2.19 on page 20 - http://www.apple.com/macbook/features.html
Figure 2.20 on page 20 - http://igesture.org/impl_introduction.html
Figure 2.21 on page 21 - http://igesture.org/impl_gesturerepresentation.html
Figure 2.22 on page 21 - http://igesture.org/impl_algorithm.html
Figure 2.23 on page 22 - http://igesture.org/impl_recogniser.html

Implementation

Figure 4.4 on page 45 - http://tuio.org/