Midas: A Declarative Multi-Touch Interaction Framework

Christophe Scholliers[1], Lode Hoste[2], Beat Signer[2] and Wolfgang De Meuter[1]
[1]Software Languages Lab
[2]Web & Information Systems Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{cfscholl,lhoste,bsigner,wdmeuter}@vub.ac.be

ABSTRACT

Over the past few years, multi-touch user interfaces emerged from research prototypes into mass market products. This evolution has been mainly driven by innovative devices such as Apple's iPhone or Microsoft's Surface tabletop computer. Unfortunately, there seems to be a lack of software engineering abstractions in existing multi-touch development frameworks. Many multi-touch applications are based on hard-coded procedural low level event processing. This leads to proprietary solutions with a lack of gesture extensibility and cross-application reusability. We present Midas, a declarative model for the definition and detection of multi-touch gestures where gestures are expressed via logical rules over a set of input facts. We highlight how our rule-based language approach leads to improvements in gesture extensibility and reusability. Last but not least, we introduce JMidas, an instantiation of Midas for the Java programming language and describe how JMidas has been applied to implement a number of innovative multi-touch gestures.

Author Keywords

multi-touch interaction, gesture framework, rule language, declarative programming

ACM Classification Keywords

D.2.11 Software Engineering: Software Architectures; H.5.2 Information Interfaces and Presentation: User Interfaces

General Terms

Algorithms, Languages

INTRODUCTION

More than 20 years after the original discussion of touch-screen based interfaces for human-computer interaction [9] and the realisation of the first multi-touch screen at Bell Labs in 1984, multi-touch interfaces have emerged from research prototypes into mass market products. Commercial solutions, including Apple's iPhone or Microsoft's Surface tabletop computer, introduced multi-touch user interfaces to a broader audience. Various manufacturers currently follow these early adopters by offering multi-touch screen-based user interfaces for their latest mobile devices. There is not only an increased use of multi-touch gestures on touch sensitive screens but also based on other input devices such as laptop touchpads. Some multi-touch input solutions are even offered as separate products like in the case of Apple's Magic Trackpad[1]. Furthermore, large multi-touch surfaces, as seen in Single Display Groupware (SDG) solutions [14], provide new forms of copresent interactions.

While multi-touch interfaces offer significant potential for an enhanced user experience, the application developer has to deal with an increased complexity in realising these new types of user interfaces. A major challenge is the recognition of different multi-touch gestures based on continuous input data streams. The intrinsic concurrent behaviour of multi-touch gestures and the scattered information from multiple gestures within a single input stream results in a complex detection process. Even the recognition of simple multi-touch gestures demands for a significant amount of work when using traditional programming languages. Furthermore, the reasoning over gestures from multiple users significantly increases the complexity. Therefore, we need a clear separation of concerns between the multi-touch application developer and the designer of new multi-touch gestures to be used within these applications. The gesture designer must be supported by a set of software engineering abstractions that go beyond simple low level input device event handling.

In software engineering, a problem can be divided into its *accidental* and *essential complexity* [1]. Accidental complexity relates to the difficulties a programmer faces due to the choice of software engineering tools. It can be reduced by selecting or developing better tools. On the other hand, essential complexity is caused by the characteristics of the problem to be solved and cannot be reduced. While the accidental complexity of today's mainstream applications is addressed by the use of high-level programming languages such as Java or C#, we have not witnessed the same software engineering support for the development of multi-touch applications. In this paper, we present novel declarative programming language constructs in order to tackle the accidental complexity of developing multi-touch gestures and to enable a developer to focus on the essential complexity.

[1]`http://www.apple.com/magictrackpad/`

TEI'11, January 22–26, 2011, Funchal, Portugal.

We start with a discussion of related work and present the required software engineering abstractions for multi-touch frameworks. We then introduce Midas, our three-layered multi-touch architecture. After describing the implementation of JMidas, a Midas instantiation for the Java programming language, we outline a set of multi-touch application prototypes that have been realised based on JMidas. A critical discussion of the presented approach and future work is followed by some general conclusions.

RELATED WORK

Recently, different multi-touch toolkits and frameworks have been developed in order to help programmers with the detection of gestures from a continuous stream of events produced by multi-touch hardware [7]. The frameworks that we are going to discuss in this section provide some basic software abstractions for recognising a fixed set of traditional multi-touch gestures, but most of them do not support the definition of new application-specific multi-touch gestures.

The *Sparsh UI* framework [11] is an open source multi-touch library that supports a set of built-in gestures including *hold*, *drag*, *multi point drag*, *zoom*, *rotate*, *spin* (two fingers hold and one drags) as well as *double tap*. The implementation of gestures based on hard-coded mathematical expressions limits the reusability of existing gestures. The framework only provides limited support to reason about the history of events. Sparsh UI provides historical data for each finger on the touch-sensitive surface by keeping track of events. As soon as a finger is lifted from the surface, the history related to the specific finger is deleted. This makes it difficult to implement multi-stroke gestures but on the other hand avoids any garbage collection issues. Furthermore, Sparsh UI does not deal with the resolution of conflicting gestures. Overall, Sparsh UI is one of the more complete multi-touch frameworks providing some basic software abstractions but offers limited support for multi-stroke and multi-user gestures.

Multi-touch for Java (*MT4j*)[2] is an open source Java framework for the rapid development of visually rich applications currently supporting *tap*, *drag*, *rotate* as well as *zoom* gestures. The architecture and implementation of MT4j is similar to Sparsh UI with two major differences: MT4j offers the functionality to define priorities among gestures but on the other hand it does not provide historical data. Whenever an event is retrieved via the TUIO protocol [6], multiple subscribed gesture implementations, called processors, try to lock the event for further processing. The idea of the priority mechanism is to assign a numeric value to each gesture. Gesture processors with a lower priority are blocked until processors with a higher priority have tried (and failed) to consume the event. With such an instantaneous priority mechanism, a processor has to decide immediately whether an individual event should be consumed or released. However, many gestures can only be detected after keeping track of multiple events, which limits the usability of the priority mechanism. Finally, the reuse of gesture detection functionality is lacking from the architectural design.

[2] `http://mt4j.org`

Grafiti [10] is a gesture recognition management framework for interactive tabletop interfaces providing similar abstractions as Sparsh UI. It is written in C# and subscribes to a TUIO input stream for any communication with different hardware devices. An automated mapping of multiple lists to multiple fingers is also not available and there are no constructs to deal with multiple users. Therefore, permutations of multi-touch input have to be performed manually which is computationally intensive and any reusability for composite gestures is lacking. Furthermore, the static time and space values are limiting the dynamic environment of multi-touch devices. The framework allows gestures to be registered and unregistered at runtime. In Grafiti, conflict resolution is based on instantaneous reasoning and there is no notion of uncertainty. The offered priority mechanism is similar to the one in MT4j where events are consumed by gestures and are no longer available for gestures with a lower priority.

The *libTISCH* [2] multi-touch library currently offers a set of fixed gestures including *drag*, *tap*, *zoom* and *rotate*. The framework maintains the state and history of events that are performed within a widget. This allows the developer to reason about event lists instead of individual events. However, multi-stroke gestures are not supported and an automatic mapping of lists to fingers is not available. Incoming events are linked to the topmost component based on their x and y coordinates. Local gesture detection is associated with a single widget element for the total duration of a gesture. A system-wide gesture detection is further supported via global gestures. Global gestures are acceptable when working on small screens like a phone screen, but these approaches cease to work when multiple users are performing collaborative gestures. The combination of local and global gestures is not supported and gestures outside the boundaries of a widget require complex ad-hoc program code.

Commercial user interface frameworks are also introducing multi-touch software abstractions. The Qt[3] cross-platform library provides gestures such as *drag*, *zoom*, *swipe* (in four directions), *tap* as well as *tap-and-hold*. To add new gestures, one has to create a new class and inherit from the `QGestureRecognizer` class. Incoming events are then fed to that class one by one as if it would have been directly attached to the hardware API. Also the *Microsoft .NET Framework*[4] offers multi-touch support since version 4.0. A simple event handler is provided together with traditional gestures such as *tap*, *drag*, *zoom* and *rotate*. In addition, Microsoft implemented a *two-finger tap*, a *press-and-hold* and a *two-finger scroll*. However, there is no support for implementing more complex customised gestures.

Gestures can also be recognised by comparing specific features of a given input to the features of previously recorded gesture samples. These so-called template-based matching solutions make use of different matching algorithms including Rubine [12], Dynamic Time Warping (DTW), neural networks or hidden Markov models. Most template-based

[3] `http://qt.nokia.com/`

[4] `http://msdn.microsoft.com/en-us/library/dd940543(VS.85).aspx`

gesture recognition solutions perform an offline gesture detection which means that the effect of the user input will only be visible after the complete gesture has been performed. Therefore, template-based approaches are not suitable for a number of multi-touch gestures (e.g. pinching). Some gesture recognition frameworks, such as iGesture [13], support template-based algorithms as well as algorithms relying on a declarative gesture description. However, these solutions currently offer no or only limited support for the continuous online processing of multi-touch gestures.

While the presented frameworks and toolkits provide specific multi-touch gesture recognition functionality that can be used by an application developer, most of them show a lack of flexibility from a software engineering point of view. In the following, we introduce the necessary software engineering abstractions that are going to be addressed by our Midas multi-touch interaction framework.

Modularisation Many existing multi-touch approaches do not modularise the implementation of gestures. Therefore, the implementation of an additional gesture requires a deep knowledge about already implemented gestures. This is a clear violation of the separation of concerns principle, one of the main principles in software engineering which dictates that different modules of code should have as little overlapping functionality as possible.

Composition It should be possible to easily compose gestures in order to define more complex gestures. For example, a *scroll* gesture could be implemented by composing two *move up* gestures.

Event Categorisation When detecting gestures, one of the problems is to categorise the events (e.g. events from a specific finger within the last 500 milliseconds). This event categorisation is usually a cumbersome and error-prone task, especially when timing is involved. Therefore, event categorisation should be offered to the programmer as a service by the underling system.

GUI-Event Correlation While the previous requirement advocates the preprocessing of events, this requirement ensures the correlation between events and GUI elements. In most of today's multi-touch frameworks, all events are transferred to the application from a single entry point. The decision about which events correlate to which GUI elements is left to the application developer or enforced by the framework. However, the reasoning about events correlating to specific graphical components should be straightforward.

Temporal and Spatial Operators Extracting meaningful information from a stream of events produced by the multi-touch hardware often involves the use of temporal and spatial operators. Therefore, the underlying framework should offer a set of temporal and spatial operators in order to keep programs concise and understandable. In current multi-touch frameworks, there is no or limited support for such operators which often leads to complex program code.

MIDAS ARCHITECTURE

The processing of input event streams in human-computer interaction is a complex task that many frameworks address by using event handlers. However, the use of event handlers has proven to violate a range of software engineering principles including composability, scalability and separation of concerns [8]. We propose a rule-based approach with spatio-temporal operators in order to minimise the accidental complexity in dealing with multi-touch interactions.

The Midas architecture consists of the three layers shown in Figure 1. The *infrastructure layer* contains the hardware bridge and translator components. Information from an input device is extracted by the hardware bridge and transferred to the translator. In order to support different devices, concrete Midas instances can have multiple hardware bridges. The translator component processes the raw input data and produces logical facts which are propagated to the fact base in the Midas *core layer*. The inference engine evaluates these facts against a set of rules.

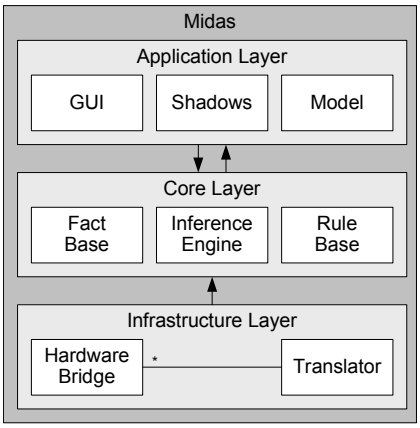

Figure 1. Midas architecture

The rules are defined in the Midas *application layer* but stored in the rule base. When a rule is triggered, it can invoke some application logic and/or generate new facts. Furthermore, GUI elements are accessible from within the reasoning engine via a special shadowing construct.

Infrastructure Layer

The implementation of the infrastructure layer takes care of all the details to address the hardware and transforming the low level input data into logical facts. A fact has a type and a number of attributes. The core fact that every Midas implementation has to support is shown in Listing 1.

Listing 1. Core fact

```
1 (Cursor (id ?id) (x ?x) (y ?y) (x−speed ?xs)
2         (y−speed ?ys) (time ?t) (state ?s))
```

This core fact has the type `Cursor` and represents a single cursor (e.g. a moving finger) from the input device. The attributes `id`, `x`, `y`, `x-speed`, `y-speed` and `time` represent the id, position, movement and time the cursor moved. The attribute `state` indicates how the cursor has changed and can be assigned the values `APPEAR`, `MOVE` or `DISAPPEAR`.

Core Layer

The Midas core layer consists of an inference engine in combination with a fact base and a rule base that is going to be described in the following.

Rules

We use rules as an expressive and powerful mechanism to implement gesture recognition. Listing 2 outlines the implementation of a simple rule that prints the location of all cursors. The part that a rule should match in order to be triggered is called its prerequisites (before the ⇒), while the actions to be performed if a rule is triggered are called its consequences.

Listing 2. Example rule

```
1 (defrule PrintCursor
2     (Cursor (x ?x) (y ?y))
3  =>
4   (printout t ``A cursor is moving at: '' ?x ``,'' ?y))
```

The first line shows the definition of a rule with the name `printCursor`. This rule specifies the matching of all facts of type `Cursor` as indicated on the second line. Upon a match of a concrete fact, the values of the two `x` and `y` attributes will be bound to the variables `?x` and `?y`. Subsequently, the rule will trigger its actions (after the ⇒) and print the text "A cursor is moving at:" followed by the `x` and `y` coordinates.

Temporal Operators

As timing is very important in the context of gesture recognition, all facts are automatically annotated with timing information. This information can be easily accessed by using the dot operation and selecting the time attribute. Midas defines a set of temporal operators to check the relationship between the timing attribute of different facts. The temporal operators and their definitions are shown in Table 1.

Operator	Args	Definition
tEqual	f1,f2	$\|f1.time - f2.time\| < \varepsilon_t$
tMeets	f1,f2	$f1.time - f2.time = \varepsilon_{tmin}$
tBefore	f1,f2	$f1.time < f2.time$
tAfter	f1,f2	$f1.time > f2.time$
tContains	f1,f2,f3	$f2.time < f1.time < f3.time$

Table 1. Temporal operators

Note that the `tEqual` operator is not defined as the absolute equality but rather as being within a very small time interval ε_t. This fuzziness has been introduced since input device events seldom occur at exactly the same time. Similar ε_{tmin}, the smallest possible time interval, is used to expresses that `f1` happened *instantaneously* after `f2`.

Spatial Operators

In addition to temporal operators, Midas offers the definition of spatial constraints over matching facts as shown in Table 2. These spatial operators expect that the facts they receive have an `x` and `y` attribute.

Operator	Args	Definition
sDistance	f1,f2	$euclidianDistance(f1, f2)$
sNear	f1,f2	$sDistance(f1, f2) < \varepsilon_s$
sNearLeftOf	f1,f2	$\varepsilon_s > (f2.x - f1.x) > 0$
sNearRightOf	f1,f2	$\varepsilon_s > (f1.x - f2.x) > 0$
sInside	f1,f2	$\beta(f1, f2)$

Table 2. Spatial operators

Again, we have a small distance value ε_s to specify that two facts are very close to each other. This distance is set as a global variable and adjustable by the developer. Since the input device coordinates are already transformed by the infrastructure layer, the value of ε_s is independent of a specific input device. For `sInside`, the fact `f2` is expected to have a `width` and `height` attribute. The β function checks whether the `x` and `y` coordinates of `f1` are within the bounding box $(f2.x, f2.y)(f2.x + f2.width, f2.y + f2.height)$. Note that we also support user-defined operators.

List Operators

The implementation of gestures often requires the reasoning over a set of events in combination with temporal and spatial constraints. Therefore, in Midas we have introduced the `ListOf` operator that enables the reasoning over a set of events within a specific time frame. An example of this construct is shown in Listing 3. The prerequisite will match a set of `Cursor` events that have the same finger id and occurred within 500 milliseconds. Finally, the matching sets are limited to those sets that contain at least 5 `Cursors`. Note that due to the declarative approach the developer does no longer have to keep track of the state of the cursors in the system and manually group them according to their id.

Listing 3. ListOf construct

```
1 ?myList <- (ListOf (Cursor (same: id)
2                            (within: 500)
3                            (min: 5))
```

A list matched by the `ListOf` operator is guaranteed to be time ordered. This is required for the multi-touch interaction domain as one wants to reason about a specific motion along the time axis. By default, rule languages do not imply a deterministic ordering but allow arbitrary pattern matching to cover all possible combinations. The spatial operators from the previous sections are also defined over lists. For example, `sAllNearBottomOf` expects two lists `l1` and `l2` and the operator will return true only if all the elements of the first list are below all elements of the second list.

Movement Operators

The result of the `ListOf` construct is a list which can be used in combination with movement operators. A movement operator verifies that a certain property holds for an entire list. For example, the *movingUp* operation is defined as follows:

$$movingUp(list) \iff \forall i, j : i < j \wedge list[i].y > list[j].y$$

In a similar way, we have defined the *movingDown*, *movingLeft* and *movingRight* operators.

Application Layer

The application layer consists of a regular program which is augmented with a set of rules in order to describe the gestures. A large part of this program however will deal with the GUI and some gestures will only make sense if they are performed on specific GUI objects. As argued in the introduction, the reasoning over GUI objects in combination with the gestures should be straightforward. Therefore, in a Midas system the GUI objects are reified as so-called *shadow facts* in the reasoning engine's working memory. This implies that the mere existence of the GUI objects automatically give rise to the associated fact.

The fields of a GUI object are automatically transformed into attributes of the shadow fact and can be accessed like any other fact fields. However, a shadow fact differs from regular facts in the sense that the values of its attributes are transparently kept synchronised with the values of the object it represents. This allows us to reason about application level entities inside the rule language (i.e. graphical objects). Moreover, from within the reasoning engine the methods of the object can be called in the consequence block of a rule. This is done by accessing the predefined `Instance` field of a shadow fact followed by the name and arguments of the method to be invoked. Listing 4 shows an example of calling the `setColor` method with the argument `"BLUE"` on a circle GUI element.

Listing 4. Method call on a shadow fact instance

```
1  (?circle.Instance setColor ''BLUE'')
```

Finally, the attributes of a shadow fact can be changed by using the *modify* construct. When the modify construct is applied to a shadow fact, the changes are automatically reflected in the shadowed object.

Priorities

When designing gestures, parts of certain gestures might overlap. For example, the gesture for a single click overlaps with the gesture for a double click. If the priority of the single click would be higher than the double click gesture, a user would never be able to perform a double click since the double click would always be recognised as two single click gestures. Therefore, it is important to ensure that the developer has means to define priorities between different gestures. In Midas, gestures with a higher priority will always be matched before gestures with a lower priority. An example of how to use priorities in rules is given in Figure 5. The use of priorities allows the programmer to tackle problems with overlapping gestures. The priority concept further increases modularisation since normally there is no intrusive code needed in order to separate overlapping gestures.

Listing 5. Priorities

```
1  (defrule PrioritisedRule
2     (declare (salience 100))
3     <prerequisites>
4     =>
5     <consequence> )
```

'Flick Left' Gesture Example

After introducing the core concepts of the Midas model, we can now explain how these concepts can be combined in order to specify simple gestures. The *Flick Left* gesture example that we are going to use has been implemented in numerous frameworks and interfaces for photo viewing applications. In those applications, users can move from one photo to the next one by flicking their finger over the photo in a horizontal motion to the left. In the following, we show how the *Flick Left* gesture can be implemented in a compact way based on Midas .

One of the descriptions of the facts representing a *Flick Left* gesture is as follows: *"an ordered list of cursor events from the same finger within a small time interval where all the events are accelerated to the left"*. The implementation of new gestures in Midas mainly consists of translating such descriptions into rules as shown in Listing 6.

Listing 6. Single finger 'Flick Left' gesture

```
1  (defrule FlickLeft
2     ?eventList[] <-
3           (ListOf (Cursor (same: id) (within: 500) (min: 5)))
4     (movingLeft ?eventList)
5     =>
6     (assert (FlickLeft (events ?eventList))))
```

The prerequisites of the rule specify that there should be a list with events generated by the same finger by making use of the `ListOf` construct. It further defines that all these events should be generated within a time frame of 500 milliseconds and that the list must contain at least 5 elements.

Gesture Composition

We have shown that set operators enable the declarative categorisation of events. The Midas framework supports temporal, spatial and motion operators to declaratively specify different gestures. Furthermore, priorities increase the modularisation and shadow facts enable the programmer to reason about their graphical objects in a declarative way.

In the Midas framework, it is common to develop complex gestures by combining multiple basic gestures as there is no difference in reasoning over simple or derived facts. This reusability and composition of gestures is achieved by asserting gesture-specific facts on gesture detection. The reuse and composition of gestures is illustrated in Listing 7, where a *Double Flick Left* composite gesture is implemented by specifying that there should be two *Flick Left* gestures at approximately the same time.

Listing 7. 'Double Flick Left' gesture

```
1  (defrule DoubleFlickLeft
2     ?upLeftFlick <- (FlickLeft)
3     ?downLeftFlick <- (FlickLeft)
4     (sAllNearBottomOf ?downLeftFlick.events
5                       ?upLeftFlick.events)
6     (tAllEqual ?upLeftFlick ?downLeftFlick)
7     =>
8     (assert (DoubleFlickLeft)))
```

The implementation of this new composite gesture in traditional programming languages would require the use of additional timers and threads. By adopting a rule-based approach, one clearly benefits from the fact that we only have to provide a description of the gestures but we do not have to be concerned on how to derive the gesture.

JMIDAS PROTOTYPE IMPLEMENTATION

We have implemented a concrete JMidas prototype of the Midas architecture and embedded it in the Java programming language. This enables programmers to define their rules in separate files and to load them from within Java. It also implies that developers can make use of Java objects from within their rules in order to adapt the GUI. We first describe which devices are currently supported by JMidas and then outline how to program these devices based on JMidas.

Input Devices

In order to support a large amount of different multi-touch devices, we have decided to implement an abstraction layer currently supported by many multi-touch devices. Two important existing abstraction layers providing a common and portable API for input device handling are the TUIO [6] and the TISCH protocol [2]. We chose TUIO as one of the input layers in the JMidas prototype implementation because of its features in terms of portability and performance. This implies that any input device supporting the TUIO protocol can automatically be used in combination with JMidas.

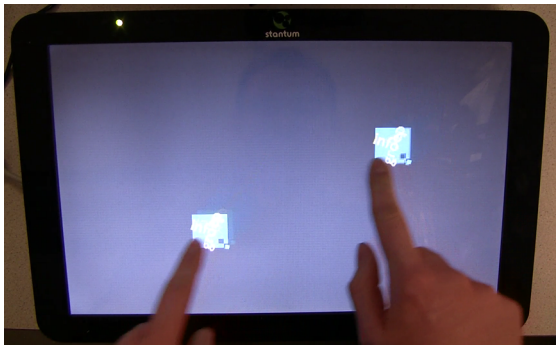

Figure 2. JMidas application on a Stantum SMK 15.4

A first multi-touch device that has been used in combination with the JMidas framework is the Stantum SMK 15.4[5] shown in Figure 2, a 15.4 inch multi-touch display that can deal with more than ten simultaneous touch points. Unfortunately, the Stantum SMK 15.4 does not offer any TUIO support and we had to implement our own cross-platform TUIO bridge that runs on Linux, Windows and Mac OS X.

Note that the Midas framework is not limited to multi-touch devices. A radically different type of input device supported by the JMidas framework are the Sun SPOTs[6] shown in Figure 3. A Sun SPOT is a small wireless sensor device that embeds an accelerometer to measure its orientation or motion.

[5] `http://www.stantum.com`
[6] `http://sunspotworld.com`

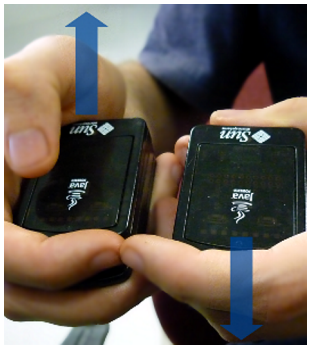

Figure 3. JMidas Sun SPOT integration

Previous experiments with the Sun SPOT device have shown that gestural movements can be successfully recognised by using a Dynamic Time Warp (DTW) template algorithm [5]. However, it was rather difficult to express concurrent gestural movements of multiple Sun SPOTs based on traditional programming languages due to similar reasons as described for the multi-touch domain. These difficulties can be overcome by using the temporal operators offered by JMidas and a developer can reason about 3D hand movements tracked by Sun SPOTs. The JMidas Sun SPOT support is still experimental but we managed to feed the reasoning engine with simple facts for individual Sun SPOTs and the recognition results for composite gestures look promising.

Initialisation

The setup of the JMidas engine is a two step process to be executed in the application layer as shown in Listing 8. First, a new JMidas engine is initialised (line 1) which corresponds to setting up the Midas core layer introduced earlier in Figure 1. The second step consist of connecting one or multiple input sources from the infrastructure layer (lines 2–8). This step also specifies a resolution for the display via the `RES_X` and `RES_Y` arguments. After this initialisation phase, the engine can be started and fed with rules. In Listing 8 this is highlighted by calling the `start` method and loading the rules from a file called `rules.md` (lines 9–10).

Listing 8. JMidas engine initialisation

```
1  JMidasEngine engine = new JMidasEngine();
2  TuioListener listener =
3      new TuioToFactListener(engine, RES_X, RES_Y);
4  TuioClient client = new TuioClient();
5  client.addTuioListener(listener);
6  SunSpotListener spListener = new SunSpotListener(engine);
7  SunSpotClient spClient = new SunSpotClient();
8  spClient.addSunSpotListner(spListener);
9  engine.start();
10 engine.loadFile(''rules.md'');
```

In JMidas, we use Java annotations to reify objects as logical facts. When a class is annotated as `Shadowed`, the JMidas framework automatically reifies the instances as logical facts with the classname as type and the attributes corresponding to the fields of the object. The field annotation `Ignore` is used to exclude specific fields from being automatically reified as attributes of the logical fact.

An example of how to use the Java annotation mechanism is given in Listing 9. Since the `Circle` object is annotated as `Shadowed`, its instances will be reified by JMidas as logical facts. The `Circle` class has the three fields `x`, `y` and `z`, whereas the last field has been annotated as `Ignore`. This implies that the `z` field will not be reified as an attribute of the logical fact. From within the logical rules, the programmer can match objects of type `Circle` by using the following expression: `(Circle (x ?x) (y ?y))`.

Listing 9. Shadowing GUI elements

```
1 public @Shadowed class Circle {
2     int x;
3     int y;
4     @Ignore
5     int z;
6     Circle(int id, int x, int y) { ... }
7 }
```

Reasoning Engine

JMidas incorporates an event-driven reasoning engine based on the Rete [3] algorithm. The core reasoning engine used in JMidas is the Jess [4] rule engine for Java which had to be slightly adapted for our purpose. The Jess reasoning engine has no built in support for temporal, spatial or motion operations. Moreover, we have extended the engine to deal with ordered list operations. Other reasoning engines could have led to the same result but we opted for the Jess reasoning engine due to its performance and the fact that it already offers basic support for shadow facts. However, we felt that in the context of our work this integration was not convenient enough and added several annotations to ensure that the synchronisation between facts and classes is handled by the framework.

Implemented Gestures

JMidas implements all of the traditional gestures such as *drag*, *tap*, *double tap*, *stretch* and *rotate*. In addition to these standard gestures, we have validated our approach by implementing a set of complex gestures shown in Figure 4. According to the Natural User Interface Group, (NUI group)[7], these complex gestures possibly offer a more ergonomic and natural way to interact with multi-touch input devices.

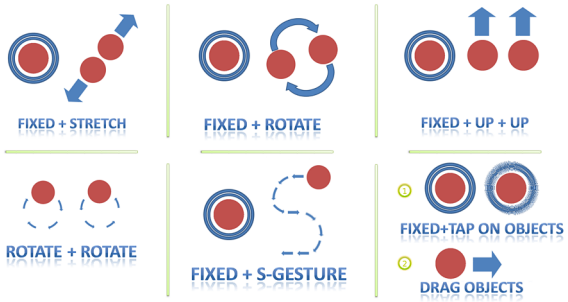

Figure 4. Complex gestures implemented in JMidas

Note that we have not empirically evaluated the claims of the NUI group since the focus of our work is to prevent the programmer from dealing with the *accidental complexity* while *implementing* multi-touch applications. To the best of our knowledge, our complex gestures shown in Figure 4 are currently not implemented by any other existing multi-touch framework.

[7] `http://nuigroup.com`

DISCUSSION AND FUTURE WORK

The implementation of gesture-based interaction based on low-level programming constructs complicates the introduction of new gestures and limits the reuse of common interaction patterns across different applications. Current multi-touch interfaces adopt an event-driven programming model resulting in complex detection code that is hard to maintain and extend. In an event-driven application, the control flow is driven by events and not by the lexical scope of the program code, making it difficult to understand these programs. The developer has to manually store and combine events generated by several event handlers. In Midas, a programmer only has to provide a declarative description of a gesture and the underlying framework takes care of how to match gestures against events.

Current state-of-the-art multi-touch toolkits provide limited software abstractions to categorise events according to specific criteria. Often they only offer information that there is an event with an identifier and a position or provide a list of events forming part of a single stroke. However, for the implementation of many multi-stroke gestures, it is crucial to categorise these events. The manual event classification complicates the event handling code significantly. With the presented Midas `ListOf` operator, we can easily extract lists of events with certain properties (e.g. events with the same id or events within a given time frame).

With existing approaches, it is difficult to maintain temporal invariants. For example, when implementing the click and double click gestures, the gesture developer has to use a timer which takes care of triggering the click gesture after a period of time if no successive events triggered the double click gesture. This results in code that is distributed over multiple event handlers and, again, significantly complicates any event handling. We have introduced declarative temporal and spatial operators to deal with this complexity. In contrast to most existing solutions, Midas further supports the flexible composition of new complex gestures by simply using other gestures as rule prerequisites.

Our Midas frameworks addresses the issues of modularisation (M), composition (C), event categorisation (EC), GUI-event correlation (G) as well as the temporal and spatial operators (TS) mentioned in the related work section. Table 3 summarises these software engineering abstractions and compares Midas with the support of these requirements in current state-of-the-art multi-touch toolkits. This comparison with related work shows that, in contrast to Midas, none of the existing multi-touch solutions addresses all the required software engineering abstractions.

As we have explained earlier, there might be conflicts between different gestures in which case priorities have to be

	M	C	EC	G	TS
Sparsh-UI	+/-	+/-	+/-	+/-	-
MT4j	+/-	-	-	+/-	-
Grafiti	+/-	+/-	+/-	+/-	-
libTISCH	+/-	-	+/-	+	-
Qt4	+/-	+/-	-	+/-	-
.NET	-	-	-	-	-
Templates	+	-	-	-	-
Midas	+	+	+	+	+

Table 3. Comparison with existing multi-touch toolkits

defined to resolve these conflicts. In current approaches, this conflict resolution code is scattered over the code of multiple gestures. This implies that the addition of a new gesture requires a deep knowledge of existing gestures and results in a limited extensibility of existing multi-touch frameworks [7]. In contrast, our Midas framework offers a built-in priority mechanism and frees the developer from having to manually implement this functionality.

In the future, we plan to develop an advanced priority mechanism that can be adapted dynamically. We are currently also investigating the JMidas integration of several new input devices, including Siftables[8] and a voice recognition system. Even though these devices significantly differ from multi-touch surfaces, we believe that the core idea of applying a reasoning engine and tightly embedding it in a host language can simplify any gesture recognition development.

CONCLUSION

We have presented Midas, a multi-touch gesture interaction framework that introduces a software engineering methodology for the implementation of gestures. While existing multi-touch frameworks support application developers by providing a set of predefined gestures (e.g. pinch or rotate), our Midas framework also enables the implementation of new and composed multi-touch gestures. The JMidas prototype highlights how a declarative rule-based language description of gestures in combination with a host language increases the extensibility and reusability of multi-touch gestures. The presented approach has been validated by implementing a number of standard multi-touch gestures as well as a set of more complex gestures that are currently not supported by any other toolkit. While the essential complexity in dealing with advanced multi-touch interfaces cannot be reduced, we offer the appropriate concepts to limit the accidental complexity. We feel confident that our approach will support the HCI community in implementing and investigating innovative multi-touch gestures that go beyond the current state-of-the-art.

REFERENCES

1. F. P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.
2. F. Echtler and G. Klinker. A Multitouch Software Architecture. In *Proc. of NordiCHI 2008, 5th Nordic Conference on Human-Computer Interaction*, pages 463–466, Lund, Sweden, 2008.
3. C. L. Forgy. RRete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
4. E. Friedman-Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications, July 2003.
5. L. Hoste. Experiments with the SunSPOT Accelerometer. Project report, Vrije Universiteit Brussel, 2009.
6. M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A Protocol for Table-Top Tangible User Interfaces. In *Proc. of GW 2005, 6th Intl. Workshop on Gesture in Human-Computer Interaction and Simulation*, Ile de Berder, France, May 2005.
7. D. Kammer, M. Keck, G. Freitag, and M. Wacker. Taxonomy and Overview of Multi-touch Frameworks: Architecture, Scope and Features. In *Proc. of Workshop on Engineering Patterns for Multi-Touch Interfaces*, Berlin, Germany, June 2010.
8. I. Maier, T. Rompf, and M. Odersky. Deprecating the Observer Pattern. Technical Report EPFL-REPORT-148043, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2010.
9. L. H. Nakatani and J. A. Rohrlich. Soft Machines: A Philosophy of User-Computer Interface Design. In *Proc. of CHI '83, ACM Conference on Human Factors in Computing Systems*, pages 19–23, Boston, USA, December 1983.
10. A. D. Nardi. Grafiti: Gesture Recognition mAnagement Framework for Interactive Tabletop Interfaces. Master's thesis, University of Pisa, 2008.
11. P. Ramanahally, S. Gilbert, T. Niedzielski, D. Velázquez, and C. Anagnost. Sparsh UI: A Multi-Touch Framework for Collaboration and Modular Gesture Recognition. In *Proc. of WINVR 2009, Conference on Innovative Virtual Reality*, pages 1–6, Chalon-sur-Saône, France, February 2009.
12. D. Rubine. Specifying Gestures by Example. In *Proc. of ACM SIGGRAPH '91, 18th Intl. Conference on Computer Graphics and Interactive Techniques*, pages 329–337, Las Vegas, USA, August 1991.
13. B. Signer, U. Kurmann, and M. C. Norrie. iGesture: A General Gesture Recognition Framework. In *Proc. of ICDAR 2007, 9th Intl. Conference on Document Analysis and Recognition*, pages 954–958, Curitiba, Brazil, September 2007.
14. J. Stewart, B. B. Bederson, and A. Druin. Single Display Groupware: A Model for Co-present Collaboration. In *Proc. of CHI '99, ACM Conference on Human Factors in Computing Systems*, pages 286–293, Pittsburgh, USA, May 1999.

[8] `http://sifteo.com`