GestureAgents: An Agent-Based Framework for Concurrent Multi-Task Multi-User Interaction

Carles F. Julià, Nicolas Earnshaw, Sergi Jordà
Music Technology Group
Universitat Pompeu Fabra, Barcelona, Spain
carles.fernandez@upf.edu earnshaw.nico@gmail.com sergi.jorda@upf.edu

ABSTRACT

While the HCI community has been putting a lot of effort on creating physical interfaces for collaboration, studying multi-user interaction dynamics and creating specific applications to support (and test) this kind of phenomena, it has not addressed the problems implied in having multiple applications sharing the same interactive space. Having an ecology of rich interactive programs sharing the same interfaces poses questions on how to deal with interaction ambiguity in a cross-application way and still allow different programmers the freedom to program rich unconstrained interaction experiences.

This paper describes GestureAgents, a framework demonstrating several techniques that can be used to coordinate different applications in order to have concurrent multi-user multi-tasking interaction and still dealing with gesture ambiguity across multiple applications.

Author Keywords

Concurrent interaction, multi-user,gesture framework, agent-exclusivity

ACM Classification Keywords

H.5.2 User Interafaces: Input devices and strategies; H.5.3 Group and organization interfaces: Computer-suported cooperative work; H.5.3 Group and organization interfaces: Synchronous interaction

INTRODUCTION

Commercial interactive general computer systems designed for multiple users have been launched before, but now, with the recent success of products like the Reactable[6] and the upcoming commercialization of Microsoft's PixelSense[1], we may anticipate that these types of devices will increase their ubiquity. The recent undeniable success of mobile computing devices with multitouch technology could boost this new wave of interaction revolution. The Mobile market can also teach us several strategies in regards to empowering developers in order to convert multi-user devices into something useful and convenient.

It has been claimed that this new type of device empowers users in several aspects or abilities, for instance it is commonly stressed that the use of tangible objects facilitates creativity and expression[1] as well as collaboration[4]. Since *tables* still constitute nowadays one of the more natural places for collaboration we expect these interfaces to unleash the programmer's creativity and allow them to exploit the software's capabilities in order to adopt the necessary interaction paradigms that allow users the most freedom. One of the key new features allowed by this technology is the capability of multi-user local concurrent multitasking, meaning that two different people could do two totally unrelated tasks at the same time and space. This is a general condition for having rich successful collaboration[7].

We must not artificially limit the capability of multi-user interaction in these kinds of interfaces, on the contrary we must support real multitasking, allowing several tasks to be performed by (probably) multiple people on the same interface. We want to avoid employing a simplistic solution like having interaction-limiting spaces such as windows; for all practical purposes this is no better than just having individual tablet devices, since it utterly limits most of the possible collaboration. We must then envisage systems that allow the applications to share the device's resources, including the physical interface, in a way that is not limiting for other apps.

An inspiration could be a table for group work. It is a shared space where users use tools; some share them while others do not. There can be several activities done at the same time, with tools used across them. A post-it note can be used on a shared map; but also on a private book. A map can be used for multiple purposes at the same time. In spite of all these potential concurrent activities, the designer or maker of the table should probably not explicitly provide support for any of them, not worrying, for example, whether a tea cup is compatible with a sheet of paper.

Imagine a Tabletop system were tangible objects and touches can be both used as input. On this Tabletop there is a music generating application like the Reactable that works by placing objects on the surface, manipulating virtual controls

[1] http://www.pixelsense.com/

TEI 2013, February 10 - 13, 2013, Barcelona, Spain.

and drawing waveforms with fingers on the surface. On this same Tabletop we also have a typical application for photo sorting that can detect camera devices on the surface and displays digital pictures on the table. Both applications potentially use the whole tabletop surface. Imagine then two people wanting to sort pictures and play music on this same Tabletop. Our vision is that one user could be grouping pictures using a lasso gesture while another could be changing the loop of one cube and muting the output of an oscillator by crossing out the waveform with a finger, without things interfering with each other. On this concurrent multi-user multitasking interface, everything can occur at the same time in the same shared space.

Gesture recognition and disambiguation in a single application

An interface or application that allows complex gesture interaction must decide the meaning of every single input event. When it finds that one same sequence of input events has several gesture candidates it has to disambiguate them.

This process can be eluded by avoiding ambiguous situations altogether, for instance by designing the application to have orthogonal gestures and inputs, which would imply that different gestures don't share interaction spaces or sources. TurTan[3] is an example of a tangible tabletop that uses pucks and touches for its interaction, separating object-related gestures and touch-related gestures into two totally unrelated sets of gestures.

Another factor used to limit the possible gesture ambiguities is context. For instance a given application can accept one type of gesture at one state and another in another state. The same can happen amongst areas of interaction: if a button is only *tappable* and a canvas is only *drawable* the application can use spatial information to rule out gestures. In this case we can think of multiplexing the interaction by time or space[2].

Finally more complex multi-gesture (multi-modal) interactions can be considered that don't use the limiting strategies defined above. There are many ways to handle the disambiguation process. Typical approaches are machine learning techniques[14, 17] or custom analytic disambiguation code, coded ad-hoc or generated by a static analysis of the possible gestures[11]. Note here that to be able to use these techniques the programmers must know at coding time all of the possible gestures that can happen at run-time.

The problem of simultaneous apps in disambiguation

As we envisage a system that allows for applications designed by third party developers, we must assume that applications developed by different people can be run and used at the same time. This poses a problem to the traditional methods of disambiguation as there may be ambiguity between gestures from different applications, which cannot be known at coding time.

The simultaneous use of unrelated applications also affects the context-based mechanisms to reduce ambiguity: multiple contexts from multiple applications have to be taken into account. Even if each application uses an *orthogonal gestures* approach, the combination of the two gesture spaces from two applications can result into ambiguity.

The sharing of space and the freedom of application developers to define gestures leads to an incompatibility which can only be corrected by limiting certain aspects of the current situation:

- We can limit concurrency by avoiding multi-tasking so only one application can be active at a time. This is mostly the current situation happening in tablets, and it invalidates our aim to allow collaboration via multi-tasking and multi-user interaction.
- We can limit the freedom of gesture definition by defining a global set of gestures and target types that apps can register to and that are recognized by the OS. This eliminates the capacity to create custom gestures, resulting in a stifling of creativity and lack of complex gestures.
- We can limit the context by forcing all of the gestures to be defined using a special restricting language created for the description of user input patterns [11, 15, 10]. This has the problem of not only restricting the influence of the application context, but also the potential complexity of the gestures, as the programmer is unable to describe a gesture beyond the limitation of this special language.
- We can limit the interaction space used by the applications by defining areas or windows where the different applications are confined, but that is exactly the situation that we want to avoid.
- As a last approach, we can limit the recognizer infrastructure, imposing some degree of collaboration. This means forcing programmers to add mechanisms for collaboration across applications in their recognizer's code. This is the approach we use, as it is the less limiting for allowing gesture complexity.

Some existing systems already use this last kind of limitation. There are special cases in mobile device systems where two applications can share the same interaction space (if we consider the operating system to be an application). For instance, in the Apple's iPad[2] there are some system gestures that can be recognized even when running applications. In these cases the disambiguation technique used could be described as *disambiguation by cascade*: first the system tries to recognize its gestures, and then the application recognizes its own. This approach, however, is only valid between apps when we can set priorities between applications (Focus), something that is justifiable for global system gestures, but not between different applications as the notion of multi-user multi-task denies interaction Focus.

In contrast, our approach simply requires developers to code their recognizers in a way that allows the system to have information about the context and the input events involved in the gestures being recognized. The system decides during the interaction whether a recognizer can or cannot identify

[2] `https://www.apple.com/ipad/`

each input event as part of a gesture. It basically defines a set of sensible rules that all recognizers must meet, but it doesn't force a particular coding style or technique.

In particular we created a framework that:

- Is device-agnostic: e.g. it doesn't matter if we are using multi-touch interfaces or depth-perceptive cameras.
- Doesn't enforce a specific programming technique or library for gesture recognizing.
- Allows concurrent gestures.
- Allows both discrete and continuous gestures.
- Allows multiple applications to share sensors and interfaces without the need of sub-surfaces or windows.
- Frees the developer from knowing the details of any other gesture that can occur at the same time and device.
- Tries to take into account real time interaction needs.

RELATED WORK

There are other frameworks for gesture recognition in multimodal interfaces such as multitouch screens that address the problem of disambiguation. For a comparison overview we suggest reading the paper from Kammer [9].

When faced with ambiguous gestures we can find that different frameworks differ on how they handle them. Some choose which gesture will "win" based on a probabilistic approach where each possible gesture is assigned a probability in a given situation, here the system will favor the most likely one. This probability could be computed using positional information of the input related to possible targets as well as completeness of the gesture[16]. Other frameworks rely on a list of priorities that the developer can define, gestures of low priority are "blocked" till gestures considered more important have already failed, this logic is present in Midas[15], mt4j[12], Grafiti[13] and in Proton[11] frameworks. Some frameworks will attempt to take a decision as soon as possible to reduce lag, others will prioritize certainty and will not resolve the conflict until there is merely one possible alternative left [15].

Proton[11], Midas[15] and GeForMT [10] allow the programmer to describe gestures in specially crafted languages that simplify the programming of recognizers. In Proton a static analysis of possible ambiguities is also performed. However, none of them have any multi-user support.

Dynamo[5] addresses a related but more specific issue: using shared surfaces (communal) to share digital data between several users. It does not, however, deal with gesture disambiguation nor third party applications in the shared space.

TDesktop[8] constituted an unpublished first attempt at creating an operating system based on a tangible tabletop platform. It did address the problem of different applications in shared interactive spaces by partially restricting the gesture vocabulary and defining virtual sensitive objects called widgets. Its approach didn't address ambiguity nor complex gestures.

BASIC CONCEPTS

Before explaining the mechanisms and inner workings of our framework, we should introduce and define some basic concepts we will be using: Agent, Gesture and Recognizer.

Agent

An agent is an entity capable of emitting events related to user input. An example of an Agent could be a sensor in an input device. The mouse position, for instance, emits updates of its relative movement. We can think of an Agent as a concept, it is not necessarily a computational object: we can define Agents at various abstraction levels. It can be a real user who emits all of the input event*s*, it can be one single finger of her, or it can be a concrete action done by one of these fingers (like a Tap event). An Agent creates a *stream of input event*s and has a beginning (creation/appearance) when it stats the stream, and an ending when it stops sending events forever.

Gesture

Given an Agent (or various Agents), that is (are) the origin of a stream of input events, a gesture can be defined by the coupling of meaning (or function) to a series of input events. For instance, the events generated by two fingers on a surface can be interpreted as a Swipe Gesture.

Recognizer

A Gesture Recognizer is a block of software that identifies gestures from a series of input events. We will often assume that for every Gesture there is a Gesture Recognizer class whose instances are dedicated to identifying different possible gestures in interaction streams coming from Agents. Take a Tap recognizer as an example, it listens to events from one Agent *finger* and tries to recognize the pattern related to a Tap.

Recognizer-Agent relation

In our framework, Recognizers are related to Agents in two ways. Firstly, as discussed, recognizers listen into interaction streams generated by agents. In order to explain the second way, consider this: since a complex gesture can be defined by a combination of several lower level gestures, its Recognizer could be simply listening to the events created by these lower level gestures instead of the original agents. We can consider that Gestures are also Agents, as they can create a stream of events over time; so in our framework the **Recognizers have Agents associated to the gestures they are recognizing**. Following our previous example, a *double-Tap* Recognizer simply listens to events from *Tap* Agents issued by the *Tap* Recognizer. It tries to recognize the pattern of a *double-Tap* from *Tap* events. The *double-Tap* Recognizer later creates and manages a *double-Tap* Agent that sends *double-Tap* events to other Recognizers or the Application.

ASSUMPTIONS AND RESTRICTIONS

As we explained in the introduction, in order to successfully disambiguate possible conflicting gestures in a shared interactive space, we have to limit some aspects of the system,

either the interaction or the programming techniques. We will explain our assumptions and limitations below. These are not simply limitations of the current implementation, but limitations by design that constitute the bedrock for our whole system.

Agent exclusivity

The first assumption in this framework for gestures is a very central one. In fact most of the disambiguation comes from this *Axiom* : **One input event can only be part of one Gesture.** This means that one Agent, at one given time, can only be assigned (or related) to one Gesture.

One concrete example would be this: one mouse button press (one input event) can only carry (or be part of) one meaning (Gesture = Agent→meaning). It may not mean both a click and a double-click. Things have been this way most of the time in existing systems like the WIMP, where the concept of Focus forces every input event to have an obvious target that identifies its meaning/gesture. For instance, Agents that have no spatial context, like keystrokes, have a target widget defined by a Focus point so a single keystroke always goes to a single target. This restriction seems also reasonable, as it is difficult to imagine attaching more than one gesture/meaning to one atomic operation in a given system .

This is, however, a limitation. Systems that are not deterministic or that for artistic purposes want to mix incompatible gesture sets without defining the priorities or policies required by this restriction, must be built in a different way and cannot benefit from this framework.

Real Time Restriction

We would like to distinguish two different types of gestures that exist when we consider the the Real-Time dimension of interaction: Discrete (or Symbolic) Gestures and Continuous Gestures (also known as Online and Offline gestures[9, 15]) . Discrete gestures are gestures that don't trigger any reaction until they are finished[9]. A typical example could be a hand writing recognition system where the user must finish the stroke before it is recognized. On the other hand Continuous gestures may trigger reactions while the gesture is still performed. For instance a pinch gesture can be applied constantly to a map while it is performed. This distinction is important not only at its implementation level, but also in a conceptual one: Discrete Gestures don't acquire meaning (and don't change the system state or trigger any kind of reaction) until they are completed. For instance, in a CLI (where only discrete Gestures exist) nothing happens until the *Enter* key is pressed.

On the other hand, continuous gestures do already convey meaning before they are completed. This means that at some point, part of their meaning is defined (typically the Type of the Gesture) while other parameters can still change, ideally in real time. In such a case we can call these parameters Controls. Our system must try to support Real Time interaction. That means:

- Supporting continuous gestures.

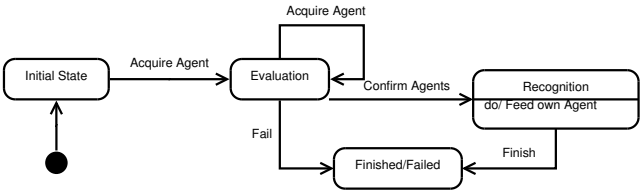

Figure 1. Main states of a Recognizer

- Avoiding unnecessary delay due to disambiguation before control is unblocked on a continuous gesture. This often implies setting a maximum waiting time on recognizers. From this we can derive the following rule: **Gesture Recognizers must try to decide whether a stream of input events can be assigned to a Gesture as soon as possible.**
- Sticking to a decision. When a continuous gesture is already identified and begins its control, switching to identifying this same agent as performing another gesture in the middle of its action could cause confusion (explicit exceptions can be defined though using policies). Imagine a user starting to move a virtual element and suddenly painting over it with the same gesture because in the middle of performing the gesture the system decided that *painting* was more appropriate than *moving*.

DESCRIPTION OF THE SYSTEM

The main idea behind our framework is to use an Agent Exclusivity logic to manage a combination of gesture sets that may be provided by different applications. These gestures are recognized by Gesture Recognizers implemented by the application developers and by independent libraries that follow a common set of rules and programming interfaces.

Life cycle of a Recognizer

Each Recognizer instance can transition through 4 main states:

Initial state the Recognizer is waiting for new Agents.

Evaluation state the Recognizer is evaluating a possible gesture which may or may not be recognized.

Recognition state the Recognizer is confident that the events match its gesture pattern, from now it simply evaluates them to recognize control parameters.

Failed/finished state the recognizer is no longer active due to a missrecognition or ending of the gesture.

The key point to understand is that Recognizers can be in a hypothetical state where they are not sure about the validity of their recognized gesture. In this state it is safe to abort the recognition as no information has been sent to its own Agent. When the gesture is confirmed and it starts sending information to it it is not possible to abort, the changes are final. By doing this it ensures the Real Time restriction of sticking to a decision.

A recognizer starts in the initial state, and it subscribes to the Agent types it is "interested" in. At this moment the Recognizer does not have any information about the gesture

it will be recognizing. When a new Agent appears, the recognizer can declare its interest in it by *acquiring* it. When a Recognizer acquires an Agent, the Agent simply adds it to a list of recognizers that are processing its events. This information is later used to enforce the Agent Exclusivity rule. When the Recognizer acquires an Agent, it creates its own Agent (which represents the gesture being recognized) and it transitions to the evaluation state. In it, the recognizer processes the input Agent events and evaluates whether they fit into its gesture pattern. As early as the Recognizer notices that these events don't match the expected gesture pattern, it *fails*. Also, as early as the Recognizer notices that the events definitively do match the expected gesture pattern, it *confirms* the Agents already acquired. When confirming the Agents, these take a decision about whether the Recognizer can have their exclusivity. This decision will depend on the presence of other Recognizers acquiring and confirming those Agents (the actual logic behind this is explained in the *Policies* section). If any of the Agents confirmed considers that the Recognizer can't have its exclusivity , it will make the Recognizer fail. Otherwise the Recognizer will move on to the Recognition state. In Recognition state the Recognizer receives the events from the confirmed input Agents and feeds events into its own agent. Whenever the Recognizer detects that the gesture is completed, it can *finish*. When a Recognizer fails or finishes, it is erased from their acquired Agents lists allowing other Recognizers to use them.

Explaining this through a simple example, a Tap Recognizer subscribes to the Touch Agent type to receive events about new Touch Agents. When a new Touch Agent is instanced, it acquires it, and subscribes to its events. The recognizer is now in the evaluation state. If the touch moves too far away from its starting position or time exceeds a limit then the agent does not match the gesture pattern of a Tap, so the Recognizer fails. When the touch finishes, our Recognizer confirms the touch Agent, as it is sure that the sequence of events is representing a tap. Then it enters the recognition state. Now it sends the touch event through its own Agent. As the tap has already been finalized, the recognizer shall *finish*.

Effects on Recognizer composition

Recognizers have associated Agents that represent the gestures being recognized. These can be used by higher-level Recognizers as input Agents, as we have seen with tap and double-tap Recognizers. When an Agent related to a Sensor (and not to a Recognizer) appears, the registered Recognizers acquire them and create their own Agents. The Recognizers subscribed to this second group of agents will then acquire them and create their own Agents in turn. This pattern can be repeated indefinitely to create a whole tree of Recognizers. As the original Agent keeps sending events, Recognizers start to match these to their gesture patterns and some will start to fail. Eventually only one branch of the original tree will remain active, only then can we consider this gesture successfully disambiguated.

Competition for Agent Exclusivity

When more than one Recognizer is interested in an Agent at a given time, they will both all acquire it. While they are in an evaluation state it is perfectly sensible for all of them to exist: they represent the several possible gestures that may be interpreted out of what this particular Agent has done so far. At some point a recognizer may try to confirm this Agent. It will then wait for the Agent to give it its Exclusivity. Typically, once all other Recognizers that acquired this Agent fail, the one that confirmed it gets its exclusivity, as it is left without any competitors. If while this first Recognizer is waiting for the Agent's exclusivity, another Recognizer confirms the Agent as well, the Agent is responsible of deciding if this new Recognizer replaces the previous as a candidate to be completed. The losing one in this competition is forced to fail. At the end all the Recognizers shall be failed by gesture mismatch or replaced in the candidacy for the exclusivity, and only one can be left. Then it is this one that obtains the exclusivity of the Agent.

Recognizer instances as Hypothesis

The previous mechanism is useful not only to disambiguate between different types of gestures (swipe gesture vs tap gestures), but also to disambiguate between different possible gestures of the same type (double tap gesture 1 vs double tap gesture 2).

Allowing concurrent multi-gesture interaction allows for having the same gesture performed in two places at the same time, which can bring difficulties. A hypothetical Recognizer that takes into account more than one of these gestures as input Agents must face the decision of whether a new Agent is part of the tracked gesture or not and acquire it in consequence. This can be problematic as the type of the new Agent may not be know it a priori. Imagine a double-tap Recognizer that has already recognized the first of its taps and is waiting for a second one, consider that before the second tap is performed another user performs a single tap somewhere else on the table as part of a completely unrelated action. In this case our double-tap Recognizer would fail after realizing this new tap is too far away to match its gesture pattern and that would be the end of it, even though the first user might have been just about to perform a second tap.

We can use the technique of competition for Agent Exclusivity to solve this type of problems. This can be done by creating a duplicate of the Recognizer instance before taking the decision of considering the new agent as part of the gesture or not. Both decisions are then evaluated, one on each instance. Eventually these two instances must compete for the exclusivity of Agents as any other pair of Recognizers. At some point the instance that took the wrong decision will fail and let the good instance win. We consider every Recognizer instance as a gesture hypothesis; every irreversible decision made generates a new hypothesis.

On our double-tap Recognizer example, when the new tap is detected a duplicate of the Recognizer will be created and only one of them will take this new agent into consideration while the other will wait for other taps. When the position of

the new tap is evaluated and identified as being too far away from the first, only one of the Recognizers shall fail.

As in this case the competition is not between two unrelated recognizers programmed by potentially different people, but between two instances of the same Recognizer the programmer can preview the obvious conflicts between the multiple instances generated by this method. For this reason an instance can ask the acquired Agent to force-fail all of the competing instances of its same type when it is sure that they are wrong.

This technique can also allow for a simple *factory-like* strategy for the use of Recognizers. At the beginning one instance of every Recognizer is created, this instance is in its initial state. As any time a new Agent is introduced we create a new hypothesis-instance, there will always be one (and only one) Recognizer instance in the initial state, ready for new Agents to be tracked while all other instances are coupled with existing Agents. It is important to mention that none of the Agents associated to Recognizers are duplicated when the presence of a new input creates a new hypothesis: An agent is not fed with events until the Recognizer confirmed its source Agents, and this does not happen until only one instance is left.

Life cycle of Agent

We already described the two main events in the life-cycle of an Agent: its beginning and its end. Let's take the example of a finger touch: the beginning of this Agent would be when the finger contacts the surface and is detected by the sensors and its end would be when the finger is removed from the surface and is no longer detected by the sensors.

But there are other milestones in the life-cycle of an Agent, mainly related to its relation to Recognizers. Recognizers can acquire, confirm and finally dismiss (upon failing) Agents. Those actions are handled internally in the Agent:

Acquiring an Agent The Agent has a list of Recognizers that have acquired it. When a Recognizer acquires the Agent, the Agent simply adds it to this list.

Confirming an Agent When this happens the Agent simply removes the Recognizer from the Acquired list and puts it to a special slot for its candidate for exclusivity. If the slot is not empty, the agent decides (via Policies) whether or not the new candidate can replace the old one, and the loser is forced to fail. At any time as from then, if the Agent (via Policies) decides that the candidate can have its exclusivity it may grant it to the recognizer.

Dismissing an Agent This happens when a Recognizer that acquired an Agent fails or finishes. If the Recognizer was in the acquired lists or in the candidate slot then it is removed from them. If it was given the Agent's exclusivity, it means that the gesture is finished. In some cases the Agent may still exist after this and other recognizers may be interested in it. Suppose for instance that a finger has drawn a circle, completing one of the existing gestures, but that it has not yet been lifted off the table. It is then treated as a new Agent but, to distinguish it from a fresh one, it is flagged as *recycled*. This way gestures may inquire if a new Agent is new or recycled, and discriminate. Most Recognizers will only be interested in fresh Agents.

Policies

Although the decision of accepting one Recognizer in favor of another when competing for exclusivity candidacy is taken inside the Agent, the actual policies are not there. In our system the Agent polls an ordered list of policy functions that either take an informed decision or delegate it to the next policy. Those policies can be defined anywhere in the code, at library level, gesture library level, custom gesture level or application level.

A simple example of this kind of policy could be "_accept_if_none" that states a preference for the new Recognizer in the candidacy if the slot is empty:

```
def _accept_if_none(current_recognizer,
                    new_recognizer):
    if current_recognizer == None:
        return True #Accept the change
```

More advanced policies can include complex metrics related to the recognizers (like "Recognizers with more acquired Agents win") or simple comparisons between Recognizer types (as "Recognizer zoom always beats Recognizer move").

The sorting in the list of policies is done with a precedence index provided when registering the policy.

Decisions requiring policies

Decisions to be taken by Agents can be defined by using two sets of policies: *completion_policy* and *compatibility_policy*.

Completion_policy is consulted when confirming an Agent. It decides whether the new candidate Recognizer to exclusivity can replace the old one in the special slot. We have already seen examples of that kind of policy.

The other set of policies, *compatibility_policy*, is used to decide whether a Recognizer can be given the exclusivity of one Agent while another one is still acquiring it. It may sound strange as it seems that here we are breaking our exclusivity rule, but in fact we are only affecting the disambiguation mechanism as we still only allow one Recognizer to use the events from this gesture. What we are in fact allowing is latent Recognizers taking over other Recognizers still in the recognizing state, as when the latent Recognizer which has acquired the Agent confirms it, it may replace the original one, forcing it to fail.

As an example we can take the translate gesture and the zoom-rotate gesture in a map browsing application: every zoom-rotate gesture may start with exactly the same sequence as the translate one. We know that it is OK that when a zoom-rotate recognizer is sure about the gesture, the translate one must fail. In this case we can create a policy stating that a zoom-rotate recognizer may be latently acquiring an Agent that is exclusively owned by the translate one:

```
def zoom_over_move(r1,r2):
    if type(r1) == RecognizerMove and
       type(r2) == RecognizerZoomRotate:
        return True
```

This kind of decentralization of this sort of decisions gives us a very large space for customization and flexibility of the system at any level: we can think of adding policies that favor Recognizers from one application over another based on the distance to an application token or any other arbitrary metric or we can add support to user identification and try to take precedence to recognizers with agents from the same user, all of that without actually changing the system's code.

Context polling

We mentioned before that context could be used for disambiguation. Instead of forcing the application to declare interaction areas or to manually activate and deactivate gestures at certain moments, we use a simple technique that can be called context polling.

Every time a gesture to which an application is subscribed issues a new Agent, the application is asked whether it is interested in it. At this moment the agent only has some initial information, but nothing forbids the Recognizer from asking again at any time when it has more complete data. When asked, the application can dismiss the Agent, and then the Recognizer shall fail. As this is done recursively, all the intermediate recognizers over which this final gesture was edified fail as well, there must always be an application interested in the final product.

An example of the benefits of this technique can be explained with a Tap and a double Tap: in contexts (areas, states...) where only a Tap is possible, the system does not have to wait until the double Tap fails in order to report the Tap; in this way the Tap is recognized faster.

An advantage of using context in this way, instead of directly coding it into the system, is that it makes the system much more flexible and independent, allowing multiple possible application management schemes: we allow using areas to delimit apps or widgets, but don't enforce doing it, as the system is agnostic. It also places the code for context evaluation where the context actually is: in the application code and not into an intermediate layer, were we should foresee all the possible aspects to take into consideration.

Link with the application

The application cannot directly subscribe to events from Agents. In order to enter into the competition for agent exclusivity it would need to acquire and confirm gestures, which it cannot do because it is not a Recognizer. The application needs a way to indirectly subscribe to the related Agents, which is done via a Fake Agent created by a special Recognizer named AppRecognizer that simply accepts any kind of input. It provides the missing piece in order for an application to enter the Recognizer competition. It also offers a special Agent type that mimics the original Agent but does not need to be acquired nor confirmed.

If we observe the full development of the Agent-Recognizer tree from the sensors to the application, we will always find an AppRecognizer just before reaching the application. Then the application only has to comply with the context polling mechanism, but not with the specific mechanics of the Recognizer/Agent relationship.

IMPLEMENTATION

A reference implementation has been done in order to evaluate the approach and to find all possible problems and corner cases. It has been programmed in Python for its ease for prototyping. The implementation comes with two examples to show how to use the framework: a painting application and a map browsing.

Three modules are provided: core system, 2D TUIO gesture basic library and tangible tabletop pygame based application environment. Only the first module is described in this paper, as the other two are just accessory to test the potentiality of the framework.

The code can be fount in the following repository: `https://bitbucket.org/chaosct/gesture-agents`

TESTS AND VALIDATION

In order to test the implementation of the framework we programed an application over it to run on a Reactable Experience tabletop[3] and put it to the stress of multi-user conditions. Validating the framework under the stress of multiple simultaneous applications is still a pending item. In this simple application a point is awarded whenever a gesture is performed and recognized, users need to collect as many points as possible within a constrained time period. The gestures used include a Tap, Double Tap, Tap Tempo (4 taps) and a variety of waveforms with different shapes and orientations. The score system is an incentive for users to perform the gestures and the time constraint is an incentive for these gestures to be performed fast and in an overlapping fashion. As a result we expect to have concurrent gestures occurring. We performed experiments on it doing repeated measures over a single group of subjects, users worked both alone and in pairs. We have analyzed application's event logs measuring the rate of concurrent interaction (CI) defined as:

$$CI = \sum_{i}^{gestures} duration(i)/InteractionTime$$

where $InteractionTime$ is the total time on where there is at least one gesture being performed.

We have found that the multi-user condition had a 11% higher concurrency ratio than the single user condition with a significance of .037, this implies that there was a meaningful difference in CI between conditions. We then conducted a questionnaire about the user's perception of how well the gestures were being identified but found no significant difference in answers between condiditons, implying that the

[3] `http://www.reactable.com/products/reactable_experience/`

framework performed just as well on the high CI condition as on the low CI one.

DISCUSSION

This framework must not be understood as a final solution to the problems that it addresses, but as a starting point for the development of a truly complete and standardized solution of the problem of concurrent multi-user multi-tasking interaction in shared spaces with third-party tools and applications. In that sense, the framework is intentionally left open in many ways, as the question of gesture priority and compatibility can be defined and tested easily in order to find the best solution in different situations.

The performed tests show that our framework can be a viable solution to the concurrent multi-user gesture recognition problem. Still, some issues require further work:

- Accessory agents: sometimes an agent can be related to a gesture, but only as a reference. For instance while doing a circle around a puck, the figure can be accessory but not core to the gesture, therefore it will not be governed by the rule of gesture exclusivity and can be shared by several gestures at the same time: imagine that a gesture that links two objects is performed at the same time as the object-circling gesture referenced above. Both gestures can have a single object as a reference point and yet not be exclusive to each other. How to solve this kind of gesture conflicts is an open question.
- As the framework has no standard set of gestures and the applications don't receive events until the disambiguation is complete, it is unclear how to implement temporary feedback to the user while a gesture is still not defined. Several approaches are possible, but we must find the most flexible one. As the application is responsible for the feedback yet the information about the hypothetical gesture is only in the recognizer (and potentially duplicated across hypotheses in several recognizers) we can envisage an additional entity that encapsulates this information and bridges it to the application until a single gesture can be clearly identified and executed.

CONCLUSION

In the coming years we will see a proliferation of technologies that can empower people to collaborate in rich shared interactive experiences. If we want to fully develop this potentiality we must allow developers to create tools that can be used in combination with other third-party software, so we can allow concurrent multitasking.

We believe that an Agent-exclusivity approach is a possible solution to this problem and can be the core of a more complete standard framework for the next generation of shared interfaces.

REFERENCES

1. A. Catala, J. Jaen, B. van Dijk, and S. Jordà. Exploring tabletops as an effective tool to foster creativity traits. In *TEI '12*, page 143, New York, New York, USA, Feb. 2012. ACM Press.
2. G. Fitzmaurice, H. Ishii, and W. A. S. Buxton. Bricks: laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 442–449. ACM Press/Addison-Wesley Publishing Co., 1995.
3. D. Gallardo, C. F. Julià, and S. Jordà. TurTan: A tangible programming language for creative exploration. In *TABLETOP 2008*, pages 89–92, 2008.
4. E. Hornecker. Getting a grip on tangible interaction: a framework on physical space and social interaction. *Proceedings of the SIGCHI conference on Human*, 2006.
5. S. Izadi, H. Brignull, and T. Rodden. Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. In *UIST'03*, pages 159–168, 2003.
6. S. Jordà, G. Geiger, M. Alonso, and M. Kaltenbrunner. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. In *TEI'07*, pages 139–146. ACM, 2007.
7. S. Jordà, C. F. Julià, and D. Gallardo. Interactive surfaces and tangibles. *XRDS*, 16(4):21–28, 2010.
8. C. F. Julià and D. Gallardo. TDesktop : Disseny i implementació d'un sistema gràfic tangible, 2007.
9. D. Kammer, M. Keck, G. Freitag, and M. Wacker. Taxonomy and Overview of Multi-touch Frameworks: Architecture, Scope and Features. *Patterns for Multi-Touch*, 2010.
10. D. Kammer, J. Wojdziak, M. Keck, R. Groh, and S. Taranko. Towards a formalization of multi-touch gestures. *ITS '10*, page 49, 2010.
11. K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: Multitouch Gestures as Regular Expressions. In *CHI 2012*, 2012.
12. U. Laufs, C. Ruff, and J. Zibuschka. MT4j - Multitouch For Java.
13. A. D. Nardi. Grafiti-Gesture Recognition mAnagement Framework for Interactive Tabletop Interfaces, 2008.
14. T. Schlömer, B. Poppinga, N. Henze, and S. Boll. Gesture recognition with a Wii controller. *TEI '08*, page 11, 2008.
15. C. Scholliers and L. Hoste. Midas: a declarative multi-touch interaction framework. In *TEI '11*, 2011.
16. J. Schwarz and S. Hudson. A framework for robust and flexible handling of inputs with uncertainty. *UIST '10*, 2010.
17. J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07*, page 159, New York, New York, USA, Oct. 2007. ACM Press.