Computer Supported Cooperative Work (CSCW) (2015) 23:79–108
DOI 10.1007/s10606-015-9218-5

Towards Concurrent Multi-Tasking in Shareable Interfaces

Carles F. Julià & Sergi Jordà
Universitat Pompeu Fabra, Roc Boronat 138 08018 Barcelona, Spain (Phone: +34935422104; E-mail: carles.fernandez@upf.edu; ; E-mail: sergi.jorda@upf.edu)

Abstract. Shareable interfaces, those that can be interacted simultaneously by several users, are a common tool used both in CSCW research and in real world applications. They tend however to lack a capability that has been traditionally relevant to the usefulness of computing systems: multi-tasking. In this paper we explain why a combination of the multi-user features of shareable interfaces and the multi-tasking capabilities of general-purpose computing, could be relevant for building useful systems, and why these features are not present today in most of the current prototypes and systems. We also discuss possible approaches for solving the problems that prevent shareable interfaces to fully support multi-tasking, and we present a novel approach based on a distributed, application-centered, content-based gesture disambiguation. We describe how an already existing framework, GestureAgents, implements this new approach, focusing on expanding the description of the relevant elements related to this problem, and conclude with some example applications and a discussion.

Keywords: Concurrent interaction, Multi-user, Shareable interfaces, Multi-tasking, Agent exclusivity

1. Introduction

Shareable interfaces are a common subject of study in the field of CSCW. Tabletops and vertical displays, for instance, are considered, in many ways, a good approach to promote collaboration, a circumstance that is valuable for solving complex tasks.

In the personal computer context (still the leading professional platform), complex task solving is often supported by the use of a combination of several unrelated software tools. However, the systems developed to study collaboration in shareable interfaces usually feature a single ad-hoc application that tries to cover all the aspects involved in the particular tested task. While this approach is valuable for studying many mechanisms of collaboration, as it constitutes a controlled environment in which the interaction dynamics can be tested, it still does not really represent existing real-world practices.

Previous experiences in other kinds of interfaces, such as PCs or hand-held devices, suggest that the real world use of new general purpose computing devices will need some kind of multi-tasking capabilities if they aim to support general and potentially complex task solving features and if, in short, they aspire to become useful to the general public. And yet, multi-tasking features in shareable interfaces may have deep differences even within single-user contexts.

This paper first analyzes the diverging relevant strengths of PCs and shared interfaces (Section 2). It then explores how these two sets of strengths are present in current shareable interface devices, and lists relevant work (Section 3). It next analyzes how multi-tasking could be implemented in big shared interfaces; addressing the specific problems this may pose (Section 4). It also presents GestureAgents, a concurrent multi-tasking interaction framework which solves the different issues exposed (Section 5). Several demo applications ant tests are exposed following in Section 6. Discussion is presented in Section 7 and finally conclusions exposed in Section 8.

2. From personal to collaborative computing

When designing collaborative computing appliances, we are in risk of losing some of the essential elements previously present in personal computing and thus ending up with a useless system. Reviewing the features that made personal computing useful to people can help us to prevent the latter from happening.

We here briefly review the qualities that contributed to make the personal computing platform successful and useful to its users. We focus on multi-tasking as a key element to such systems, and an element that we argue, is often overlooked in current collaborative computing systems.

We also focus on the key deficiency of personal computers, which collaborative computing intends to solve: collaboration, and more specifically multi-user interaction with computer systems. We think that the combination of these two key elements, collaboration and multi-user interaction, can bring collaborative computing system to a state of usefulness ready to be adopted.

2.1. The rise of the PC

The advent of computers and personal computing devices has had a very deep impact in recent history. Their usefulness relies in their capacity of performing tasks that previously were inconvenient, difficult or impossible, and the core goal of Human-Computer Interaction is indeed devoted to support users accomplishing these tasks (Shaer and Hornecker 2010).

Many qualities of computers contribute indeed to this goal of easing task solving processes:

- Computing power: This was the original function of computers, the ability to compute mathematical operations in an unprecedented speed. There is no need to even try to enumerate the infinity of practical applications that computing power has created, transforming our world.
- Convenience: Computers make it easy to perform simple but repetitive tasks, instantly. Boring tasks that may not be especially difficult, when automated, can be performed in a fraction of the time originally needed, thus making them accessible to much more people. Spreadsheets are a good

example: accounting existed before computers, but it is now more widely accessible.

- Connectivity: It could be argued that portable computing devices do not specially excel in computing power. Still, their usefulness is undeniable, probably because of their connectivity, which allows users to communicate in efficient ways.
- General-purpose computing: What makes computers even more useful is that they are generic tools. The purpose and function of a computer can be changed by changing its program. This uncoupling of the device from its function, frees computer builders from having to think about every possible use of the machine. Other parties can create programs that will turn that computer into different specific tools such as a word processor or a calculator. In the smartphone revolution the availability of third-party applications has also arguably been instrumental to their success.

2.2. Multi-tasking

General-purpose computing allows a single computing device to change its function according to a program. The computer itself has no specific objective as a calculator would have. Logical programs are executed by the computer instead, making it useful for a specific calculation or any task completion. On the other side, as the universe of possible complex tasks and problems to be solved with the assistance of a computer is broad and open, it seems rather unpractical to create a single program for every different task. As complex tasks can often be divided in simpler subtasks, the particularity of every task will often require the use of several different more generic programs, which will address some of the subtasks we can divide the original problem in. Those programs, on its turn, can then be reused for other different tasks.

Let's imagine, for instance, that someone is writing a report on the discoveries of new wildlife in a country. This task will require writing, editing and formatting text, capturing, classifying and editing images, calculating statistics and displaying charts, creating and manipulating maps, etc. Instead of having a single program for "new wildlife finding report writing" involving all these activities, several programs addressing the needs of every single activity can be used: a word processor, an image editor, a file browser, a spreadsheet editor, a map browser, etc. These programs, such as text editors or image viewers, usually designed to solve domain-specific problems, can be also created by parties that do not relate to the creators of the hardware or the programmers of the OS. These third-parties can be programmers or teams that have an expert knowledge of the field the program is focused on. Allowing third party software to be created without the prior consent of the computer, the OS designers or other software creators, allows new programs to continuously appear for filling potential new needs.

There is indeed a common agreement that allowing third-party apps is an important factor for success on commercialization of computing platforms. A classic example would be the effect of commercialization of the Lotus 1-2-3 spreadsheet program exclusively for the IBM's PC. Sales of IBM's PC had been slow until 1-2-3 was made public, and then increased rapidly a few months after Lotus 1-2-3′s release.[1] As a more contemporary example, Apple trademarked the slogan "there is an App for that" for its iPhone 3 g selling campaign on 2009,[2] advertising the availability of third-party apps as its main appeal. This move by Apple revolutionized the smartphone scene (West and Mace 2010).

Modern operating systems and computers allow several programs to be run in parallel, and to switch interaction with the user at any desired time. This ability, multi-tasking, helps to use computers to solve a particular task that involves several steps and requires potentially different programs, in a more convenient way than having to stop the current program to start another. Multi-tasking would be indeed very convenient in our hypothetical new wildlife finding report writing activity: while our user is writing the report, she has the need of inserting a picture of a new specimen. She switches the interaction from the word processor to a file browser to find the picture she wants, she then opens it inside an image editing program (another switch), where she crops the marginal part of the picture. She then switches again to the word processor (that still holds the document she was working on) to insert the modified image.

We can thus conclude that multi-tasking is a very desirable feature when creating computing systems, as it provides a way to deal with real world tasks in a convenient fashion, by allowing the use of third party programs.

2.3. Multi-user

Collaboration has also been traditionally tied to complex task completion: group meetings are a common strategy to shed light into difficult problems. Big problems can be divided into smaller ones that can be redistributed (Strauss 1985; Schmidt and Liam 1992), and points of view can be exchanged (Hornecker and Buur 2006). Even in the computer era, the practice of physical meetings seems to be still (if not more than ever) prevailing. Empowering collaboration with computers is the primary goal of Computer Supported Collaborative Work (CSCW) field, and it relates directly to this group meeting problem. In this discipline, two different (but intersecting) problems are studied: in co-located CSCW all group individuals are present in the same workspace while in remote CSCW individuals are located in different places and all personal interaction is mediated by computers. Both problems deal with

[1] https://en.wikipedia.org/wiki/Killer_application

[2] http://www.trademarkia.com/theres-an-app-for-that-77980556.html

several users interacting with the same (local or distributed) system, leading to multi-user interaction.

Non co-located settings for multi-user interaction in a single virtual workspace, such as web-based collaborative systems (Bowie et al. 2011), or general cases of collective distributed work on single documents (groupware) (Ellis and Gibbs 1989), are very common and widely studied. Co-located collaboration around computers, on its turn, already exists in a daily basis. Work meetings are often complemented with laptops, tablets, smartphones and other computing devices.

Needless to say, desktop and laptop computers have not been designed for co-located multi-user interaction, but for individual usage. Since they feature a single keyboard and a single pointing device, when used in multi-user setups computers inevitably lead to an interaction "bottleneck" with the users (Stanton et al. 2001; Shaer and Hornecker 2010). The use of computers in this context is thus still individual, lacking the social affordances that can be provided by "shareable" interfaces, or systems that have specifically been designed for co-located collaboration (Marshall et al. 2007). Affordances, which according to scholars such as Hornecker and Buur, should particularly consider Spatial interaction and Embodied Facilitation (Hornecker and Buur 2006).

Shareable interfaces on their side, alleviate the interaction "bottleneck" by creating multiple interaction points, preventing individuals from taking over control of the computing device (Hornecker and Buur 2006). Multiple interaction points do also promote user participation, lowering thresholds for shy people (Hornecker and Buur 2006), and can provide means for bi-manual interaction promoting a richer gesture vocabulary (Fitzmaurice et al. 1995, 1996). A typical type of interfaces developed for these collaborative scenarios are tabletop interfaces, which allow users to interact with horizontal displays using touch and/or pucks; vertical interactive displays (such as interactive whiteboards) in which users interact using pens or touch; tangibles which allow users to interact with physically-embedded artifacts and tokens (Rogers et al. 2009; Shaer and Hornecker 2010); or body gestural interfaces, such as camera-based systems, which allow users to interact using their bodies (Shaer and Hornecker 2010). As a distinct characteristic, all these interfaces allow users a shared access to the same input and output physical interfaces, as opposed to typical groupware systems, where each user has its own interface device (Rogers et al. 2009). Besides collaboration, these shareable interfaces show also affordances more directly related with complex task completion. Epistemic actions, physical constraints, and tangible representations of a problem may contribute to problem solving and planning (Shaer and Hornecker 2010). Spatial multiplexing allows for a more direct and fast interaction (Fitzmaurice George 1996) while leveraging the cognitive load (Shaer and Hornecker 2010); tangible objects facilitate creativity (Catalá et al. 2012); and rich gestures lighten cognitive load and help in the thinking process while taking advantage of kinesthetic memory (Shaer and Hornecker 2010).

This combination of social and personal affordances suggest that shareable interfaces are indeed well suited for complex task completion: apart from promoting

collaboration, they provide individual and collective benefits that help completing these goals.

3. Multi-tasking in shareable interfaces: current situation and related research

Despite all the aforementioned affordances, and considering all the multi-task desirable properties, the majority of the currently available shareable interface systems created for research purposes, consist of a single program that already includes all the necessary facilities to cover every subtask of the main activity. This is, however, consistent with the purpose of most research, because, in a collaboration co-located setting, CSCW researchers typically focus their investigations on the human factors in multi-user interaction, such as how input devices can be more effectively distributed between users in order to optimize group dynamics (Kim and Snow 2013; Verma et al. 2013), or on studying different strategies to access digital and physical items from the perspective of digital content sharing (Verma et al. 2013), control sharing (Jordà et al. 2010; Kim and Snow 2013), or proxemics (Ballendat et al. 2010).

A similar enclosing phenomenon happens with real-world products using shareable interfaces. While some of them focus on a very specific domain, avoiding to address more general problems (e.g., the Reactable (Jordà 2008) addresses collaboration from the very specific and peculiar needs of musical collaboration (Xambó et al. 2013)), many others, such as interactive whiteboards, desist about using any particular multi-user interaction, thus directly presenting the PC graphical system (Beauchamp 2004); and when addressing multi-tasking, they are single-tasked, or simply present methods to change the full-screen single active application (Ackad Christopher et al. 2010).

However, having multi-tasking capabilities in shareable interfaces seems to be in strict consonance with their goal of promoting and enabling collaborative work, as, for instance, the recommendations by Scott et al. for collaborative tabletops (Scott Stacey et al. 2003) are related: Multi-tasking provides a way to have simultaneous activities, allowing the transition between them (*support fluid transitions between activities*) and between personal and collective ones (*support transitions between personal and group work*). Also, several tasks can be done concurrently, by several users (*support simultaneous user actions*).

It would therefore seem clear that real world shareable interfaces should at least support some of the characteristics that have turned the personal computer into such a valuable tool, such as general purpose computing (and third party application support) and multi-tasking, which sadly are not yet typically found on most current research prototypes.

We argue that the lack of those features may not be an accident, neither an unconscious omission: the combination of multi-tasking -a feature so closely associated with single-user devices- with multi-user interaction, is not trivial; even less when combined with rich interfaces such as the ones provided by tabletops. And yet,

we want to stress our vision that real world collaborative systems should allow third party applications (programs) to run and be interacted simultaneously. More precisely, every program should support multi-user input, and a single user should be able to interact with several applications at the same time. This is not a novel or revolutionary idea and some works have in fact, previously attempted at the creation of multi-user multi-task systems.

Dynamo (Izadi et al. 2003), proposes a shared surface for sharing digital data between several (remote) users, focusing on ownership and permissions over programs and documents in a shared multiuser WIMP system. Users may use pairs of mouse-keyboard to interact with a system that presents local and shared interfaces. In shared interfaces it focus the attention on methods for preserving and sharing control over applications and files. It does not, however, deal with co-located access to the interface, nor with third party applications in the shared space.

LACOME (Mackenzie et al. 2012) also depicts a common shared surface in which remote single-user PC systems are presented as manipulable windows. Third party applications are allowed, but those run in the logic of the former single-user systems. A similar concept is developed in TablePortal (AlAgha et al. 2010), where remote tabletop applications and activity is presented inside manipulable windows. In this case remote applications are multi-touch enabled, although its aim is to be used by a single user, the teacher of a classroom.

(Ballendat et al. 2010) presents us with a series of devices, one of them a vertical shareable interface, which uses information such as the relative positions and orientations of the users, the devices, and other objects, and specifically their pairwise distances (proxemics), for affecting the interaction. As this information is shared between all the devices (as an Ubicomp ecology), and each device can run a different program, we could consider this example as a shared interface (based on the relative positions and orientations) with multi-tasking. The proposal does not describe however any strategy for coordinating the different programs, but rather assumes that they are created together as parts of the same system.

WebSurface (Tuddenham et al. 2009) presents a tabletop system with virtual windows that can be freely manipulated. These windows are web browsers presenting conventional web pages that can be interacted by the users. It could be argued that web pages are a form of third-party applications, although enclosed in a single-user paradigm. This is also the case of Xplane (Gaggi and Regazzo 2013), a software layer presenting several tiled windows on the surface with a distinct focus to enable fast development of tabletop applications, although it does not provide window transformation abilities.

Multi Pointer X (MPX) (Hutterer and Thomas Bruce 2007) tries to transform PCs into shared systems by allowing them to use several pairs of keyboard and mouse. As PCs are already multi-task and third-party application enabled, the result would be a shared multi-user multi-tasking system. Using a PC setting and applications, however, does not help to easily allow multi-user interaction inside the applications, neither collaboration dynamics related to the physical layout of the interfaces.

Julià and Gallardo's TDesktop (Julià and Gallardo 2007) was a first unpublished attempt to create a tabletop operating system. It provided facilities for third-party tabletop applications to be developed, as well as an environment to run and manage multiple applications at the same time. Applications were multi-user by default, and they could ask the system for full-screen execution, when not designed as floating widgets. However, it did not enforce that input events were distributed to one application at most, leaving the possibility to multiple interpretations.

In the next section we will study and try to overcome some of the technical and conceptual difficulties for designing a proper multi-tasking system on a shareable interface.

4. Approaches to multi-tasking

From an implementation point of view, interaction in multi-tasking can be narrowed down to two different problems: (i) allowing two or more processes to share the input and (ii) allowing two or more processes to share the output. In the PC, input would consist of mouse and keyboard events, whereas the output would take place in the monitor display (and in the speakers). In a tabletop system the output would also be the visual and audible display, whereas the input would be provoked by the objects and the finger touches on its surface.

Although sharing input and sharing output may be superficially seen as two aspects of the same problem they are fundamentally different. Sharing output is a relatively simple issue because it can be reduced to a mixing mechanism: many programs may require to output some data to a specific destination (the screen), and the task of such a system would simply consist on deciding how to (or rather whether to) mix these data. As the source and destination of the output events is known, the system can use simple rules to decide, for instance, if an app can draw into the display, occluding other programs, or if the sound that it is generating will be mixed with the sounds coming from other programs, and with which volume.

On the other hand, sharing input is a much more complicated de-mixing problem: data from one source (such as the data coming from the touch sensor on a multi-touch display) can potentially relate to several recipients, the programs. The task of the system on this case is more complex: the system must know the destination of every data element, that can be shared or not. On a PC, a *media play* keystroke, for instance, has to be distributed to the correct program that is waiting for these types of events, and not always necessarily to the "active" program, the one that is considered to be actually used by the user, with a privileged situation that makes it the default receiver of all input data.

4.1. Input sharing

As many programs can be potential receivers of this data, the system needs a set of rules and mechanisms to fully determine the correct recipient of every piece of input data. These rules will determine the way multi-tasking is presented to the user.

Several rules exist, for dealing with this uncertainty. Many of them will take into account the context. In interfaces where input and output happen at the same place (i.e., with Full or Nearby embodiment, according to Fishkin taxonomy (Fishkin 2004)), such as in a touchscreen, input events can be tied to the output elements nearby. A touch can be tied to the visual element just underneath it, created by a particular program that will become its correct recipient.

Interfaces in which input and output are decoupled (Sharlin et al. 2004) may impose more difficulties. When input information is completely untied to the output elements of the processes, strategies other than using a simple distance criterion have to be used. In the case of a mouse device, for instance, the PC strategy is to create a virtual pointer that is controlled by it: as this pointer is coupled to the display it can be treated as in the previous case (coupled). The mouse mediates between the user and the cursor; it is not a generic input device which is part of the interface, but a specific physical representation of the cursor.

The PC keyboard is another decoupled interface, and keyboard events can have several destinations, these being different programs or even different widgets inside a program. Some windowing systems simply send the keyboard events to the program under the pointer, while others create a default destination for the keystrokes (Scheifler and Gettys 1990). This destination is controlled by the input keyboard focus, so that only one widget (from one application -the active one) is the current receiver of all keyboard activity, and this destination can be changed using the pointer (interacting with another window/widget) or special key combinations (such as Alt-Tab in the PC). The assumption of a single input keyboard focus by the PC interaction makes it difficult to adapt it into a multi-user setting, as the interaction would require multiple foci. Some approaches have been taken in this direction, such as the Multi Pointer X (MPX) (Hutterer and Thomas 2007) extension, which allows having virtual input pairs of visual pointers and keyboards that can operate at the same time both with adapted and with legacy X11 applications. It struggles with applications that assume that there is only one pointer and focus, enforcing single-user interaction with those applications as a partial solution. By pairing cursors and keyboards in pairs, MPX allows several foci (one per pair) to simultaneously exist (Hutterer and Thomas 2008).

The approach to follow on shareable interfaces will depend on the type of interface and its purpose. Coupled input/output interfaces, such as tabletops or vertical displays have the possibility of tying input events to output entities. Gestural body interfaces may have to use other approaches, such as using a mediating virtual representation of the body (equivalent to the cursor) as the seminal work of Myron Krueger in Videoplace or Videodesk (Krueger et al. 1985) already suggested, or some other kind of focus mechanism.

Input sharing in tabletops is still a young question, as it seems that the problem of multi-tasking has still not arisen. Window-based application management is starting to be present on tables (Tuddenham et al. 2009; AlAgha et al. 2010) but the preferred option continues to be full screen locking.

4.2. Area-based interaction

In coupled interfaces it is common to find window-based multi-tasking, so that different programs obtain independent rectangular areas. They can draw and get all the interaction performed inside. Those areas can usually be transformed and manipulated by the user, making it possible for multiple processes to be present in the display at the same time, thus promoting multi-tasking. In these cases, all the programs inputs and outputs are confined inside their respective (or multiple) windows, and a simple coordinate test helps input events to be assigned to the correct program.

Rectangular windows particularly fit the PC setting. They have the same shape as the screen, and as they cannot be rotated they can occupy the full screen if necessary, occluding other windows (rotation of windows is not desirable, as the display is vertical and has a well-defined orientation, similarly to what would happen to a painting in a wall).

Using windows on other non-PC situations can have some caveats. In non-rectangular interfaces, such as in round tabletops like the Reactable (Jordà 2008), the rectangular shape seems to perform poorly. The Reactable's circular surface was designed to avoid dominant positions (Vernier et al. 2002; Jordà et al. 2005). While, perhaps for this same reason, the original Reactable avoided the use of windows or rectangular areas, its more recent commercial incarnations make use of them, and allows users to reorient them,[3] suggesting that when no predefined orientation exists, the potential rotation of windows seems necessary. Even within rectangular tabletops, at least two (or even four) predominant points of view could exist, making the rotation of windows a desired feature.

On top of these orientation issues, forcing a fixed shape for all applications may not always be a convenient solution: some programs may need less restricted areas, leaving most of its window space empty (for instance a circular program such a clock would have considerable empty space at the edges of the window). This empty space would prevent input events to reach other occluded applications, making them unreachable (see Figure 1).

4.3. Arbitrary shape area-based interaction

An alternative to window-based interaction is area-based interaction. In this case, instead of windows, the system will have to maintain a list of active arbitrary-shape areas of the processes. The input events distribution mechanism should be equivalent as when using windows: a collision test will find the correct program that holds the target area for one particular event. By using arbitrary shapes instead of rectangular windows, processes no longer have the problem of empty occlusion, as all the unused application space does not have to be covered by an area. Using arbitrary-shape areas

[3] https://www.youtube.com/watch?v=kYyg-wVYvbo

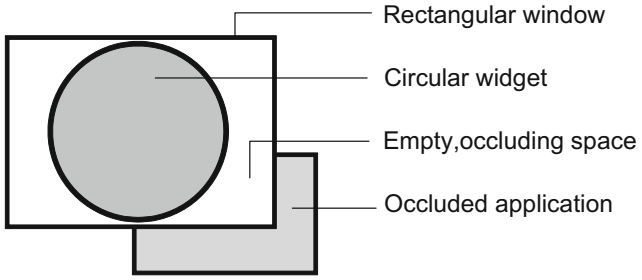

Figure 1. A window with empty space occluding the interaction for another.

is already a popular approach when distributing events through different objects inside an application. Inside a program window, the different presented elements define areas where the forwarded input event can be assigned to. Buttons, sliders and many kinds of controls are examples of this strategy.

However, this approach is not perfect. Apart from the case of decoupled interfaces, where area-based interaction is not possible, this strategy may not be desirable in other additional situations, at least as the only discriminating mechanism.

Recent history of interaction in touch-enabled devices has shown that there is room for improvement beyond the simple gesture primitives that were associated with pointing devices, and a variety of touch-based gestures have been developed and even patented since the first portable multi-touch devices appeared (e.g., pinch zoom, swipe, swipe from outside of the screen, etc.) (Hotelling et al. 2004; Elias et al. 2007).

The fact that portable devices tend to have full-screen applications, which can therefore trivially manage all the multi-touch input, has boosted the development of complementary and often idiosyncratic gestures, able to handle more complex and richer interaction. If areas were used to know the destination of every input event, the gestures of every application should start, continue and end inside of the process' areas, rendering many gestures that used to temporarily transit outside the target area, impossible to recognize. Even a strategy where only the starting event is used to check the colliding area may have problems with gestures starting outside of it. Let's imagine and study some examples of gestures that would be problematic when using areas. An application is responsible for displaying notes through the surface of a tabletop. Those notes can be translated and transformed by standard direct manipulation gestures such as pinch zoom or dragging. Imagine that the programmer wants to implement a gesture to save this note: circling the note.

Note that for circling a widget with one finger, we do not need to enter in contact with the widget itself (see Figure 2). If the area of the widget is defined by the surface of the note, the needed input events will never reach its right destination. Having a larger gesture area covering the places where gestures are likely to occur may help to receive such events, but at the cost of occluding the interaction with other event recipients, such as other potential applications underneath this note's area.

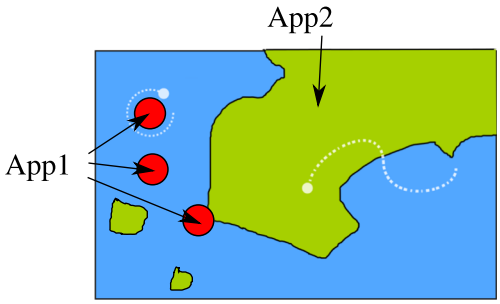

Figure 2. A note-taking application that allows the user to create new notes by drawing circles with a finger over a map browsing application that can be dragged with the finger. Notice that if using area-based interaction, the note taking program will not receive circling events.

In this other example, let's imagine a gesture (e.g., a cross) that instantiates a new widget (e.g., a new note in our note-taking application), anywhere on the interactive surface. As there is no predefined existing area listening for events, the note-taking application cannot know when and where to invoke a new note, and, if the whole-surface area was used for catching all potential crosses, other applications would be occluded and being unable to receive any input event. Although this particular example could be solved by showing a button widget to create new notes, it would have to always be visible, cluttering the space. Global system gestures could be another example of gestures made outside areas, a gesture defined by the system to show a global menu, such as a wave gesture, can be performed anywhere on the surface, regardless of whatever is underneath.

Julià and Gallardo's TDesktop (Julià and Gallardo 2007) tabletop operating system solved this problem by allowing the several applications that could run simultaneously to receive the raw stream of input events as an addition from its standard area-based input event filtering, thus receiving also input data that originated elsewhere of their areas. This solution, although effective, rises the problem of how to distribute events through applications, so to avoid the problem of having several subscriber programs receiving the same events, and each of them simultaneously assuming being the intended addressee of the interaction.

Finally, the area-based strategy to multi-task interaction is not possible with decoupled interfaces such as full-body sensors and camera-based interfaces (e.g., Kinect), motion sensors (e.g., wiimote), voice and sound interfaces (e.g., Speech recognition), these could still benefit from multi-tasking abilities as they are already used in multi-user contexts. If multi-tasking with decoupled interfaces may still seem like a fringe problem, an example can quickly reveal its need. When multiple home appliances in the same room, such as a hi-fi sound system and an air conditioner, can accept body gestures as commands, they are in fact sharing the same input interface (the body). Some mechanism has to ensure that the same body movement cannot be interpreted as commands for both appliances simultaneously.

In short, shareable interfaces (as we have seen in the example of TDesktop) trying to process area-less gestures, but also decoupled interfaces, would benefit from a mechanism different than using areas or windows, for distributing input data to its correct destination, and thus preventing various programs to process the same events.

4.4. Content/semantics -based input sharing

For decoupled systems that cannot use window (or area)-based input sharing, as well as for coupled interfaces that for some reason would opt for not using it, an alternative can be using content-based input sharing.

In a content-based input sharing mechanism, the algorithm, instead of distributing the input events to their destinations based on the position of the event, would try to know which events are expected by every application, and would then distribute these events by deducing their right destination. This approach would not necessarily treat input data as separated events, but rather as streams of events that may convey meaning within them. The destination of an input event, for instance, may not only depend on its own information, but also on the gesture it is part of, on the types and characteristics of the possible recipients, the context, etc. Generally speaking, when a series of input events that have a global meaning/semantics as a gesture is defined, the system's function is to successfully recognize the performed gesture and subsequently distribute it into the processes, given their current expectations and their contexts. A very simple example implementing this idea could be a system which has the code to recognize a set of gestures from the input, and when it fully recognizes a gesture, this is distributed to the application that has requested it. In the possible case that applications *A* and *B* request respectively the *stick* and *pinch* gestures, when the system recognizes a *stick* gesture it handles it to *A*. Instead, when a *pinch* gesture is recognized this one is sent to *B*.

An issue arises when implementing a system that uses content-based input sharing: does the system incorporate all the code needed to recognize all the defined gestures? Should the full set of gestures be defined within the system or should they be defined within the addressees programs themselves? Depending on how we choose to distribute the role of defining and recognizing these gestures, three different strategies can be employed:

a. A centralized gesture recognition engine, with a fixed set of gestures.
   As in our *stick* and *pinch gestures* example, the system could define a fixed set of gestures the applications could register to. Based on the preferences of the applications at the time a gesture is recognized, the system just notifies the correct program when an individual gesture is recognized. Unfortunately, this strategy has a clear drawback since it prevents programs to define their own gestures, the ones that the application programmer(s) felt were best suited. Rich interaction, understood as the possibility for individual applications to define their own optimal gestures independently of the existing system gestures, is thus dangerously limited.

b. A centralized gesture recognition engine, with an application-defined set of gestures.

In this type of systems, common recognizing mechanism needs to be implemented, for which the application programmers will define their own respective recognizable gestures. Many recent advances have been attained in the direction of language-based gesture definitions, especially in the context of multi-touch applications, which in our case could allow arbitrary gesture definitions to be added to the system at runtime:

Proton (Kin et al. 2012), Midas (Scholliers et al. 2011), GeForMT (Kammer et al. 2010b) and GISpL (Echtler and Butz 2012) all allow the programmer to describe gestures in specially crafted languages that simplify the programming of gesture recognizers, and therefore the code dedicated to detect gestures from the input event streams. From those, Midas, GeForMT and GISpL are interpreted (GISpL only partially) and could theoretically be used as the basis for more general systems, on wich the applications carry their own gesture definitions and transfer such specifications to the system, which would use them to recognize the gestures.

The choice of the gesture definition language is also a non-trivial issue. Such a language should ideally be as complete as possible in order not to become an obstacle for the programmers, thus making some gestures impossible to define. For instance, for allowing gestures to be related to the application context data, such as virtual objects inside the application, the definition language should provide ways to access it. Proton, GeForMT and GISpL explicitly integrate areas (as parameters to be accessed in the language or as a previous filtering) as part of their languages, easing area-based gestures to be programmed, but making area-less gestures difficult to describe, as this is a fundamental part of these languages. Midas allows instead for a sort of generic user-defined code and object access from inside the gesture definition, thus enabling not only areas, but also other types of constrains to be used, showing its potential to be useful in many gesture recognition styles. However, it is unclear how such relationship would work when applied on a server–client schema, which would need to interpret the definitions within the system while the needed code and data resides on the program. Apart from these language issues, a gesture recognizer system should also meet some additional requirements. None of the aforementioned languages allow multiple instances of gestures being performed at the same time, treating instead all the input events as part of the same gesture, thus making them unsuitable for multi-user contexts.

Although a variation of the previous projects would probably fit the requirements for building this type of system, forcing all the programs to describe their gestures in a common language would also have the side effect of preventing other kinds of gesture-recognition approaches from being used. For instance, machine learning based approaches (such as (Wobbrock et al. 2007) or (Caramiaux and Tanaka 2013)) would not be possible, since within this strategy, gestures are not formally described, but learned instead from examples.

c. A decentralized application-centered gesture recognition, with a coordination protocol

With this third strategy, the system does not participate directly on the recognition of the gestures, but helps instead in coordinating the set of programs interested in these gestures. The recognition process takes therefore place inside the applications, allowing nearly total freedom to the programmer, while a common protocol between the system and the programs is used to guarantee that no single event is mistakenly delivered to two different processes.

By running the gesture recognition inside the application, it can take into account its context (e.g., position of the application elements, and other internal logic) without having to rely on a good gesture language definition, as in the previous case. This approach also allows programmers to code the recognizers using their favorite techniques or frameworks, instead of having to rely on the system's choice of language or libraries. Furthermore, as the system is in charge of preventing double interpretations of gestures across different applications, the different recognizing mechanisms will not need to provide multi-tasking facilities. The aforementioned gesture description languages could be easily adapted to support the coordination protocol with the system, and they could be deployed inside the application. Other programs could for example use a machine learning approach provided that they respect the protocol, and thus train their gesture recognizers with examples.

The framework we are presenting, GestureAgents (Julià et al. 2013), tries to create this common protocol and infrastructure. In GestureAgents, instead of relying in the use of a particular declarative language, the recognizing mechanism is conditioned by a series of coordination messages that the system and the processes need to exchange.

5. Implementation of GestureAgents framework

In this section, we describe how the GestureAgents framework implements the proposed protocol strategy to manage input events to be consumed by recognizers implemented in several applications. We first introduce the basic elements, then the protocol between the applications and the system, the restrictions of the gesture recognizers' behavior, and give details about the functioning of the system and the particular implementation of GestureAgents.

5.1. Elements of GestureAgents

GestureAgents is a framework that aims to provide a generic and flexible solution for multi-user interaction in shareable interfaces, both inside a single application as in a multi-tasking system. As schematized in Figure 3, GestureAgents relies upon the

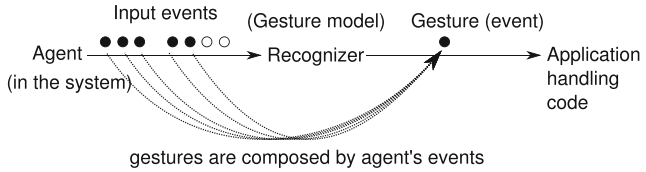

Figure 3. Conceptual elements of GestureAgents.

concepts of "agent", "gestures" and "gesture recognizers" and on the idea of "agent exclusivity".

An agent is the source component of part of the interface input events, such as an object in contact with a tangible tabletop interface or a finger touching the surface on a touch-based interface. For example, in the case of a multi-touch interface, an agent would be created for every sequence of touches, considering that a sequence starts with the detection of a finger hitting the surface and concludes when the finger is removed from the surface. By default agents will represent the minimal set of identifiable event types (such as the finger touch already described) while more high level agents, those composed by other agents, such as a hand agent composed by finger ones, can be also provided, which are best suited for full body interfaces, where the interaction can have different "resolutions".

Gestures are sequences of agents' events, which convey meaning expressed by the user and defined by the program. A gesture can relate to a single agent or to multiple ones, both simultaneously and distributed in time. Gestures can be discrete (or symbolic), in which case they will not trigger any reaction until they are finished, or continuous, which already convey meaning before they are completed, and can therefore trigger reactions before finished (Kammer et al. 2010a).

A gesture recognizer, a piece of code that checks that the pattern that defines the gesture corresponds to the received events, is used by the program to identify a gesture coming from the agents' events. By using agents as the basis for its gestures, recognizers do not have to receive all the events from the interface, but only the agents they are interested in. This allows the recognition of multiple gestures at the same time (see Figure 4), as opposed to the majority of gesture frameworks, where all the input events are part of the same gesture.

The fundamental idea in GestureAgents is based upon *agent exclusivity*. An agent, at one given time, can only be part of one gesture (see Figure 5). The system presents the input data, in the form of agents, to the gesture recognizers inside the applications, and, if they want to use them as a part of their associated gestures, they will have to compete between them to earn the exclusivity over the agent's use before recognizing their gesture. By locking different agents, several recognizers can simultaneously recognize gestures, preventing double interpretation of the same input events, and allowing multi-tasking and multi-user gesture interaction.

As GestureAgents does not use a special gestural description language, problems concerning the limits of this framework in terms of completeness or design assumptions

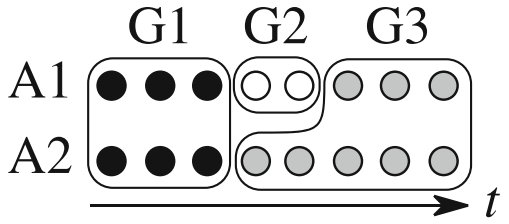

Figure 4. In this example the events, emitted by two agents (A1,A2) and represented by circles, are part of three different gestures (G1,G2,G3) that can occur simultaneously, as in the case of G2 and G3.

do not apply. Definition of gestures is done solely on the applications. Areas, if present, are also implemented at the application level, and tested by the applications' own gesture recognizers, using their own settings. It is thus up to the programmer to use any existing library to recognize gestures or to code a recognizer from scratch.

5.2. GestureAgents protocol

The coordination protocol is defined by communication between recognizers (inside applications) and agents (in the system), relating to the process of soliciting agents, getting their exclusivity and releasing them.

The communication regarding the recognition of gestures, happens between recognizers (inside the applications) and the system (holding the agents), as shown in Figure 6. The GestureAgents' protocol defines various types of relationships between recognizers and agents, depending on their internal state. Specifically, a recognizer is considered to follow a process of four distinct steps:

- Initial state
  The recognizer is waiting for an agent (of one specific type) to be announced by the system. While in this situation, the recognizer can be considered *dormant* (that it is not related to any active agent or gesture).
- Evaluation state
  The recognizer, which has communicated to the system an interest on one or several agents, is evaluating if their events match a possible gesture, which may or may not be recognized at the end. In this state, the confidence of the

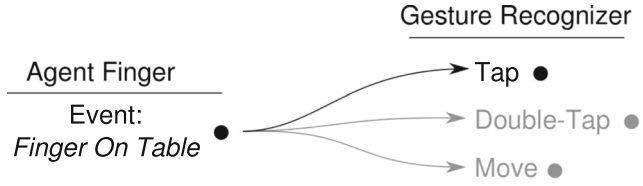

Figure 5. Agent exclusivity enforces that an agent at a given time can only be part of one single gesture.

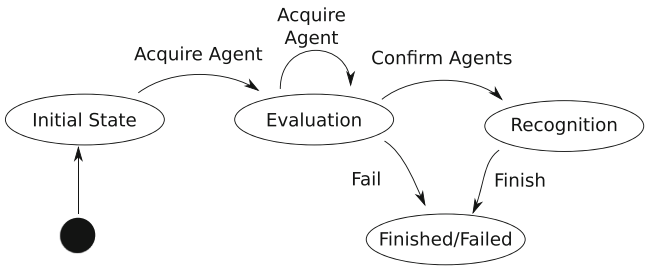

Figure 6. States of a recognizer.

recognizer for the hypothesized gesture is not high enough for considering it to be correct or incorrect.

Depending on the type of gesture being evaluated, this state can be more or less extended in time. Discrete (or symbolic) gestures will be processed mostly in this state, because their correctness is not fully set until the end of the gesture (Kammer et al. 2010b). Continuous gestures, however, can be recognized way before the gesture has ended. In this former case, this state will last as long as the type of the gesture is not confirmed.

- Recognition state

  In this phase the recognizer is confident that the tracked events of the agents match its associated gesture pattern. The transition to this state occurs after two subsequent factors: (i) the recognizer no longer considers the gesture an hypothesis (and so it abandons its evaluation state), and (ii) the system grants the recognizer the exclusivity on the requested agents. In this state the recognizer simply processes the agents' events to extract control events from the gesture, until the recognizer considers it to have ended.
- Failed/finished state

  In this state the recognizer is no longer active; this can be due to the nonrecognition of the gesture, or to the successful conclusion of the recognized gesture.

With this behavior in mind, the protocol is composed of a series of messages that can be exchanged between the recognizer and the system. From the recognizer perspective these would be the messages sendable to the system:

- Register (or unregister) to a type of agent

  If a recognizer is registered to a type of agent (for instance a "touch agent"), when a new agent of this kind appears in the system, the recognizer is notified. This message will typically happen in the recognizer's initial state.
- Register (or unregister) to an agent's event type

  Given an agent, the recognizer subscribes to its events. For instance given a touch agent it could be possible to register to its update events (movement, or pressure). In the evaluation state, the recognizer will subscribe or unsubscribe to different type of the agent's events, depending of the pattern of the associated gesture.

- Acquire an Agent (preventing other recognizers of getting its exclusivity)
  By acquiring an agent, the recognizer expresses its interest on it, communicating the system that it is currently evaluating if the agent is part of a given gesture. This message will be responded by the system with the result of the operation: *true* for success acquiring the agent; *false* for failure acquiring it. This prevents the agent to be assigned to other recognizers (from another program, for instance) until this recognizer dismisses it (due to conclusion or to nonrecognition). The recognizer will typically acquire agents in the evaluation state.
- Confirm an Agent (requesting the Agent exclusivity)
  After successfully acquiring an agent and checking for its events, the recognizer may conclude that it is part of the expected gesture. It then proceeds to confirm it. This message will only be issued by the recognizer when attempting to transition from the evaluation state to the recognition state. The response from the system may not be immediate (we will later address disambiguation delay), and until then the recognizer remains in the evaluation state. If the exclusivity is finally granted by the system, the recognizer will receive a message from the system notifying so. If the system does not grant the exclusivity, it will send a message forcing the recognizer to fail.
- Dismissing an Agent
  An agent can be dismissed in order to be reclaimed by the system, for being assigned to other recognizers. This may happen when a recognizer voluntarily considers that an acquired agent is not part of the expected gesture, or when confirmed agents are part of a gesture that the recognizer considers finalized. Also, when a recognizer fails, all the acquired and confirmed agents are forcefully dismissed.

The system will send signals to the recognizer, both (i) in response to its requests, (ii) in the case of acquiring an agent and, on its own prerogative, (iii) for notifying the presence of new agents, (iv) for transmitting agents' events, (v) for granting the exclusivity over an agent, or (vi) for forcing the recognizer to fail.

To illustrate how this protocol works we will detail a possible example of a recognizer's life-cycle, based on a recognizer that implements the recognition of the gesture "straight line over a widget" in a tabletop system, as represented in Figure 7.

In this example, when the application starts, the recognizer is instantiated by the application and it starts in its initial dormant state. It then subscribes to the *touch agent* type to receive new agents' announcements. Each time the system notifies the recognizer of the presence of a new touch agent, the recognizer checks that this agent is near a widget, as its gesture should be related to one of them. If the touch agent happens to be near a widget, the recognizer declares its interest in the agent by *acquiring* it, and entering into its evaluation state.

If this agent is not yet assigned in exclusivity to any other recognizer, the system accepts the query and communicates it to the recognizer, which subscribes to this agent's movement events, in order to track the trajectory of the

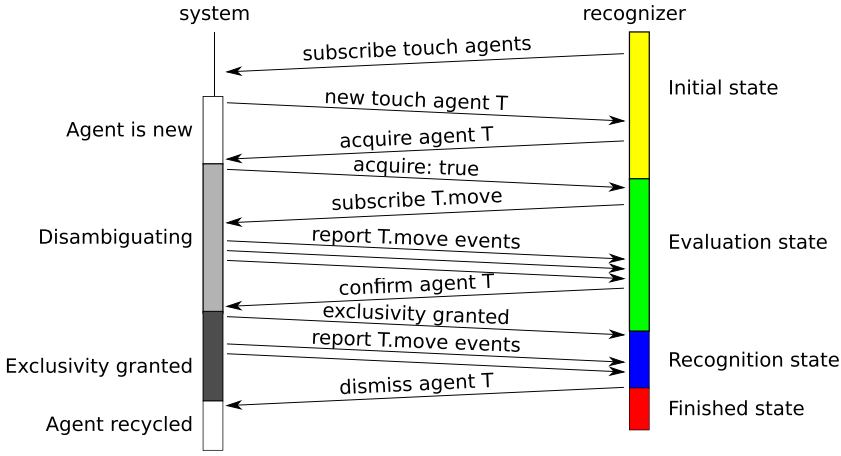

Figure 7. Protocol representation of the "straight line over a widget" gesture recognition example.

touch. While the touch agent slides through the surface, the system sends the corresponding agent events related to this movement. With every update, the recognizer keeps checking if the overall movement is indeed a straight line, and if it is crossing the widget nearby.

When the touch crosses the widget, the recognizer notices that the events definitively do match its expected gesture pattern, and it confirms the already acquired touch agent, thus requesting its exclusivity. If at this moment no other recognizer is acquiring it, the system confirms the exclusivity to the recognizer. With this confirmation, the recognizer moves to the recognition state, and starts receiving the events from the touch agent. When the recognizer decides that the gesture is completed, it finishes by dismissing the agent in the process.

5.3. Restrictions on the behaviors of recognizers

The good functioning of the described protocol depends on the recognizers implementing the protocol correctly, but also on respecting some good practices. In particular, during all the time one recognizer stays in its evaluating state it is preventing other (possibly correct) recognizers to get the agents exclusivity and enter their own recognition states.

An ill-coded recognizer, for instance, could just acquire all the agents in the system, and never fail or confirm them. This would indefinitely prevent all other recognizers to successfully earn the agents' exclusivity and thus no recognizer would ever actually recognize their corresponding gestures.

To minimize the disambiguation delay between the recognizer confirming the agents and getting their exclusivity (pictured in Figure 8), recognizers must decide as soon as possible whether a stream of input events can be or not be assigned to a gesture, thus minimizing their stay in the evaluation state.

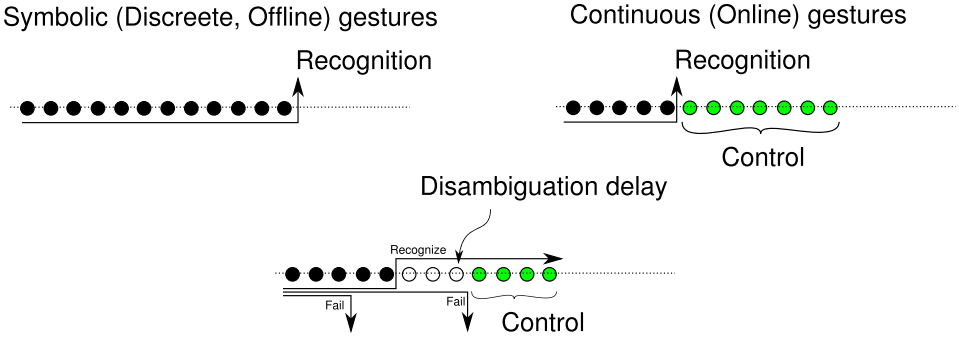

Figure 8. Disambiguation delay occurs between the evaluation state (*black*) and the recognition state (*green*).

Another consequence of this recognition process protocol is that confirming agents is a final decision. Once a recognizer enters the recognition state, the gesture should always be valid, and if the agent's events are no longer considered part of the gesture, the recognizer should finish and release all the agents' exclusivity. The agents can then be used again by other recognizers, in the condition of *recycled* agents, as their appearance is caused by the release form a previous recognizer, instead of being new.

5.4. The GestureAgents system

The rationale behind these messages is embedded in the functioning of the system while protecting the agent exclusivity. For each agent, the system manages a list of acquiring recognizers (those that are interested in the agent) and a slot for only one completing recognizer (that considers this agent as part of its gesture). When a recognizer acquires the agent, the system simply adds it to this list, unless this agent's exclusivity is already given.

When a recognizer confirms an agent requesting its exclusivity, the system removes the recognizer from the acquired list and puts it into the completing slot. If the slot is not empty, the system decides (via the consultation of several policies) whether or not the new candidate should replace the old one, and the loser (whichever it is) is forced to fail. In general, exclusivity is granted only when the list of acquiring recognizers is empty, which usually happens when alternative acquiring recognizers fail recognizing the gesture and thus dismiss the agent, removing them from the agent's list of acquiring recognizers.

When a recognizer dismisses an agent of which it had its exclusivity, this agent can be used again by other recognizers; the system sets a flag marking it as "recycled" and notifies other interested recognizers as if it was a brand new agent (an overall picture of the states and transitions of an agent is shown in Figure 9).

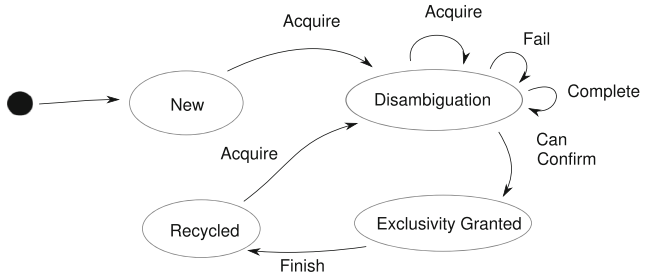

Figure 9. Life cycle of an agent.

This mechanism actually prevents agents from being used as a part of a gesture, until no other recognizers are interested. When two competing recognizers are sure that an agent is part of their gesture, a decision has to be made. Policies, an ordered list of specific rules that apply to specific situations, will deal with cases of conflict, defining priorities and compatibilities between recognizers.

The decisions to be taken by the system can be defined by using two sets of policies, completion_policies and compatibility_policies:

- The first are consulted when confirming an agent. They decide whether the new recognizer candidate for exclusivity can replace the old one in the completing slot, defining a priority between two competing recognizers. For instance, a system could decide that recognizers from applications with a given priority, will win over non-prioritized ones, or an application could enforce that a pinch zoom recognizer always wins a drag move recognize
- The compatibility_policies are used to decide whether a recognizer can be given the exclusivity over one agent, while another one is still acquiring it. Although at a first glance this may seem as if we were breaking the exclusivity rule, we are in fact only affecting the disambiguation mechanism, as we will still only allow one of the recognizer to use the events from this gesture. What this mechanism is in fact allowing is having recognizers in a "latent" state, which will allow other recognizers to use the agent until they can confirm it, thus finally provoking the recognized gesture to end. Compatibility_policies thus permit defining a priority between a confirmed recognizer and "latent" aspirants.

A common generic policy set that could be added to a system using GestureAgents, would be one prioritizing complex gestures over simple gestures. For instance, in a tabletop, prioritizing gestures involving multiple fingers over gestures involving one single finger. If we measure this complexity by the number of acquired agents, it would be simple to define a completion_policy guaranteeing that complex gesture recognizers will be granted the agents' exclusivity whenever they successfully recognize a gesture, in spite of the less complex gesture recognizers acquiring them:

```
@Agent.completion_policy.rule(0)
def complex_beat_simple(r1, r2):
    if len(r1._agentsAcquired) < len(r2._agentsAcquired):
        return True
```

By defining a similar compatibility_policy, we would allow simpler gestures to be recognized until a more complex gesture gets the exclusivity. This pair of polices would also solve the previously mentioned pinch-zoom versus drag-move gesture problem.

```
@Agent.compatibility_policy.rule(0)
def simple_can_recognize_until_complex(r1, r2):
    if len(r1._agentsAcquired) < len(r2._agentsAcquired):
        return True
```

At this point, it has to be noted that the current implementation is not using yet a real, portable network protocol, but is instead prototyped as a relationship between Python objects inside the system and the application. However it follows this pattern closely. In the current prototype implementation, policies can be defined at many levels, and can be introduced by applications, recognizers or the system itself. In a more conservative implementation, with a network-based coordination protocol, it could be more interesting that system-wide policies would only be defined inside the system, thus preventing arbitrary code from injected application-defined policies to be executed by the system. Application-based policies could be instead enforced at the application level.

5.5. The GestureAgents recognition framework

Apart from the agent exclusivity coordination protocol for multi-user and multi-touch interaction, GestureAgents provides a gesture recognition framework based on the same agent exclusivity concept. It provides gesture composition (i.e., describing a gesture in terms of a combination of previously defined simpler ones) by stacking layers of agent-recognizer relations, and by considering that recognized gestures can also be agents (such as double-tap agents). The framework also takes advantage of the agent exclusivity competition between recognizers for solving internal disambiguation for simultaneous instance of the same gesture recognizer, by treating them as different gestures that have to compete for the agent's exclusivity.

Recent developments in the framework have simplified the first layer of agent-recognizer relation, the one of the system-recognizer communication. By encapsulating every recognizer relationship tree inside an isolating proxy, the protocol becomes much clearer and eliminates possible incompatibility issues due to the use

of the compositing feature of the gesture-recognition framework. In the previous structure, there was no distinction between end-user gestures and sub-gestures.

6. Example applications and systems created with GestureAgents

GestureAgents has been used in several systems and applications, testing several aspects of the framework: a concurrency test application, a painting system demo, a map-browsing demo and an orchestra conducting simulator. Unless stated, the examples have been implemented in a Reactable Experience tabletop device.[4]

To test the performance on a multi-user condition, a gesture-performing game has been implemented. The gestures used include Tap, Double Tap, Tap Tempo (4 taps) and a variety of waveforms with different shapes and orientations. Users earn points by performing the correct gesture when asked to. Experiments done in this system show that it is capable of successfully supporting concurrent gesture recognition and interaction (Julià et al. 2013).

A painting system constituted by two separate applications has also been created to test both the agent exclusivity competition by recognizers, and the effects of the recognition delay (see Figure 10, right). One application has recognizers for the tap, stick (straight line) and paint (free movement) gestures, while another uses a double-tap recognizer in a circular area. The results of the gestures of the first application are reflected in visual elements (lines, dots and traces), while the second application erases the display when a double tap is detected. As the double tap is only valid in a circular area, performing a single tap inside the area would, at first, activate also the double-tap recognizer, to end failing after a timeout call. This setting allowed to observe that the recognition delay introduced by the double-tap happened only inside the area.

A map application, featuring typical pinch zoom and drag move gestures for manipulating a world map, as well as tap and stick gestures for annotating geographical locations and reseting the view respectively, has been created to test the different policies (see Figure 10, left). The relationship between the pinch zoom and the drag move recognizers require the first to be able to overcome the agents completed by the second, thus defining both a compatibility_policy and a completion_policy to achieve the effect.

Finally, a fully "decoupled interface" application, consisting of an orchestra conductor simulator for the detection of conductor movements using a depth camera, is currently in development (Gómez et al. 2013) (see Figure 11). The use of skeleton-based agents as the basis for the gesture recognition is helping us to clarify how multi-level agents, such as joints, limbs and users, should be used in a decoupled level without affecting the agent exclusivity competition between recognizers.

[4] http://www.reactable.com/products/reactable_experience/

Figure 10. A map browsing application (*left*) and a painting system (*right*) implemented in GestureAgents.

The GestureAgents framework is open source and available to anyone for use and improve. The code can be found in the following repository: https://bitbucket.org/chaosct/gesture-agents, and videos of some of the examples can be found at http://carles.fjulia.name/gestureagentsvideos .

7. Discussion

The GestureAgents approach to provide multi-tasking to shareable interfaces is still in a prototype stage and can primarily serve as a starting point to explore this type of application-centric distributed gesture recognition strategy. This means that many aspects regarding the real world usage of such mechanism are still to be explored and discussed in depth.

A typical concern of such system could be its resilience against ill-behaved programs. An application that unintentionally grabs input events without releasing

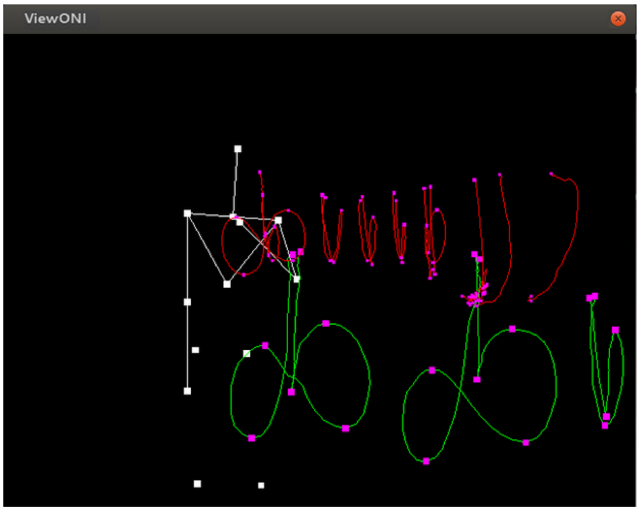

Figure 11. Orchestra conductor gesture recognizer application implemented in GestureAgents.

them, could effectively block all other programs from receiving the exclusivity over the agents to fully recognize their gestures, unless specific policies preventing or limiting this type of behavior were implemented.

Even malware could register similar or identical gestures to the ones from legit programs in order to *steal* those to insert its malicious content. Again, careful policies would have to be designed to limit this kind of attacks, such as using proximity to areas to prioritize conflicting gestures. However, the experience with PC malware tells us that it is very difficult to be protected from malicious applications.

Another security-related concern is whether an application could steal secrets from our interaction with other programs. As in GestureAgents every process can receive all input information to check if it fits a particular gesture, it is sensible to think that key-logger-like applications could be effectively developed. Being the situation similar to the PC's in this case, we can learn from its implemented strategies to solve that particular problem. Some operating systems implement a way to interact with a specific dialog that is isolated from all the other processes in order to enter a password or to confirm an action that requires specific privileges, a possible solution in GestureAgents could be based on this same idea.

Other issues related to efficiency could be relevant. The GestureAgents strategy simply distributes the events to the applications, leaving to them the recognition. In this perspective it does not pose any relevant computing burden. Additionally, the restrictions imposed on the recognizers' behavior favors incremental gesture recognition approaches, which are computationally cheap. In fact, the informal experience through the different exposed tests and demos, does not clearly reveal any perceptually relevant impact by GestureAgents.

That said, in current systems, input events are either processed in a central engine before distributing them to the applications, or are filtered by area (or by focus point) before being processed inside the application. In GestureAgents many applications can be processing the same events at the same time, multiplying the needed processing power. At least with the current implementation, this effect seems inevitable.

Existing centralized gesture recognition engines that recognize several hypothetical gestures simultaneously are making efforts to parallelize this processing while guaranteeing soft real-time (Marr et al. 2014). In GestureAgents the processing of gestures in different applications would be done in parallel by definition, although without real-time guarantees.

Overall, we think that the identification of the problem of the lack and need of multi-user concurrent multitasking, and our approach to the solution contribute to the current state of the art. By proposing a content-based disambiguation instead of an area-based one, GestureAgents approach can be a valid solution for multi-tasking, in both coupled and decoupled shareable interfaces, revealing itself as a generic solution. This can be increasingly relevant for new upcoming decoupled interfaces such as hand tracking sensors or depth cameras, which could benefit from policies and strategies developed for other more popular interfaces.

8. Conclusions

We have identified and exposed the need of multi-tasking capabilities in shareable multi-user interfaces. We have argued about the utility of multi-tasking when solving complex tasks with computers, and showed that multi-tasking features are currently missing in actual shareable interfaces, despite the fact that one of their main goals is complex task solving through collaboration between users.

We argue that this lack is not unintentional but a consequence of the difficulty of adapting current multi-tasking-capable systems into shareable interfaces.

An analysis of the complexities of implementing such a system together with a discussion of possible strategies has been carried, revealing that "area-based input events distribution" or "gesture language definition-based" approaches may pose problems in the context of rich interaction and decoupled interfaces. A third approach, using a protocol to control input event distribution but leaving gesture recognition to the application has been described and considered as the best choice.

An implementation of this approach, GestureAgents, has been presented as a possible solution, which implements the third of these strategies.

Examples of use of the framework have been finally presented, showing some of the possibilities of multi-user multi-tasking interaction and the potential of the framework itself.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7 / 2007–2013 through PHENICX project under grant agreement n° 601166.

References

Ackad, Christopher James, Anthony Collins, Judy Kay (2010). Switch: exploring the design of application and configuration switching at tabletops. *ITS'10: ACM Int. Conf. Interact. Tabletops Surfaces. Saarbrücken, Germany.* New York: ACM Press, pp. 95–104.

AlAgha, Iyad, Andrew Hatch, Linxiao Ma, Liz Burd (2010). Towards a teacher-centric approach for multi-touch surfaces in classrooms. *ITS'10: ACM Int. Conf. Interact. Tabletops Surfaces. Saarbrücken, Germany.* New York: ACM Press, pp. 187–196.

Ballendat, Till, Nicolai Marquardt, Saul Greenberg (2010). Proxemic interaction. *ITS'10: ACM Int. Conf. Interact. Tabletops Surfaces. Saarbrücken, Germany.* New York: ACM Press, pp. 121–130.

Beauchamp, Gary (2004). Teacher use of the interactive whiteboard in primary schools: towards an effective transition framework. *Technology, Pedagogy and Education*, vol. 13, no. 3, pp. 327–348.

Bowie, Muriel, Oliver Schmid, Agnes Lisowska Masson, Béat Hirsbrunner (2011). Web-based multipointer interaction on shared displays. *CSCW'11: Proc. ACM 2011 Conf. Comput. Support. Coop. Work. Hangzhou, China*. New York: ACM Press, pp. 609–612.

Caramiaux, Baptiste, Atau Tanaka (2013). Machine Learning of Musical Gestures. *NIME 2013: Proc. 2013 Conf. New Interfaces Music. Expr.* Daejeon & Seoul, pp. 27–30.

Catalá, Alejandro, Javier Jaen, Betsy van Dijk, Sergi Jordà (2012). Exploring tabletops as an effective tool to foster creativity traits. *TEI'12: Proc. Sixth Int. Conf. Tangible, Embed. Embodied Interact. Kingston, Ontario, Canada*. New York: ACM Press, pp. 143–150.

Echtler, Florian, Andreas Butz (2012). GISpL: Gestures Made Easy. TEI'12: Proc. Sixth Int. Conf. Tangible, Embed. Embodied Interact. Kingston, Ontario, Canada. New York: ACM Press, pp. 233–240.

Elias, John Greer, Wayne Carl Westerman, Myra Mary Haggerty (2007). Multi-touch gesture dictionary. US Patent 7,840,912 B2.

Ellis, Clarence A, Simon J Gibbs (1989). Concurrency control in groupware systems. *SIGMOD'89: Proc. 1989 ACM SIGMOD Int. Conf. Manag. Data. Seattle, Washington, USA*. pp. 399–407.

Fishkin, Kenneth P (2004). A taxonomy for and analysis of tangible interfaces. *Personal and Ubiquitous Computing*, vol. 8, no. 5, pp. 347–358.

Fitzmaurice, George W (1996). *Graspable user interfaces*. Ph.D. dissertation. University of Toronto: Graduate Department of Computer Science.

Fitzmaurice, George W, Hiroshi Ishii, William AS Buxton (1995). Bricks: laying the foundations for graspable user interfaces. *CHI'95: Proc. SIGCHI Conf. Hum. Factors Comput. Syst. Denver, Colorado, USA*. New York:ACM Press/Addison-Wesley Publishing Co., pp. 442–449.

Gaggi, Ombretta, Marco Regazzo (2013). An environment for fast development of tabletop applications. *ITS'13: Proc. 2013 ACM Int. Conf. Interact. tabletops surfaces. St. Andrews, United Kingdom*. New York: ACM Press, pp. 413–416.

Gómez, Emilia, Maarten Grachten, Alan Hanjalic, et al. (2013). PHENICX: Performances as Highly Enriched aNd Interactive Concert Experiences. Open access

Hornecker, Eva, Jacob Buur (2006). Getting a grip on tangible interaction: a framework on physical space and social interaction. *Proc. CHI'06: SIGCHI Conf. Hum. Factors Comput. Syst. Montréal, Québec, Canada*, New York: ACM Press, pp. 437–446.

Hotelling, Steve, Joshua A Strickon, Brian Q Huppi, et al. (2004). Gestures for touch sensitive input devices. US Patent 8,479,122 B2.

Hutterer, Peter, Bruce H Thomas (2007). Groupware support in the windowing system. *Eighth Australas. User Interface Conf. Ballarat, Australia*. Australian Computer Society, Inc., pp. 39–46.

Hutterer, Peter, Bruce H Thomas (2008). Enabling co-located ad-hoc collaboration on shared displays. *Ninth Australas. User Interface Conf. Wollongong, NSW, Australia*. Australian Computer Society, Inc., pp. 43–50.

Izadi, Shahram, Harry Brignull, Tom Rodden, et al. (2003). Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. *UIST'03: Proc. 16th Annu. ACM Symp. User interface Softw. Technol. Vancouver, BC, Canada*. pp. 159–168.

Jordà, Sergi (2008). On Stage: the Reactable and other Musical Tangibles go Real. *International Journal of Arts and Technology*, vol. 1, no. 3/4, pp. 268–287.

Jordà, Sergi, Martin Kaltenbrunner, Günter Geiger, Ross Bencina (2005). The reactable*. *ICMC 2005: Proc. Int. Comput. Music Conf. Barcelona, Spain*. pp. 579–582.

Jordà, Sergi, Carles F Julià, Daniel Gallardo (2010). Interactive surfaces and tangibles. *XRDS: Crossroads, The ACM Magazine for Students*, vol. 16, no. 4, pp. 21–28.

Julià, Carles F, Daniel Gallardo (2007). *TDesktop?: Disseny i implementació d'un sistema gràfic tangible*. Degree thesis. Universitat Pompeu Fabra.

Julià, Carles F, Nicolas Earnshaw, Sergi Jorda (2013). GestureAgents: an agent-based framework for concurrent multi-task multiuser interaction. *Proc. 7th Int. Conf. Tangible, Embed. Embodied Interact. Barcelona, Spain*. pp. 207–214.

Kammer, Dietrich, Georg Freitag, Mandy Keck, Markus Wacker (2010a). Taxonomy and Overview of Multi-touch Frameworks: Architecture, Scope and Features. *Workshop Eng. Patterns Multitouch Interfaces*

Kammer, Dietrich, Jan Wojdziak, Mandy Keck, et al. (2010b). Towards a formalization of multi-touch gestures. *ITS'10: ACM Int. Conf. Interact. Tabletops Surfaces. Saarbrücken, Germany*. New York: ACM Press, pp. 49–58.

Kim, Henna, Sara Snow (2013). Collaboration on a large-scale, multi-touch display: asynchronous interaction and multiple-input use. *CSCW'13. San Antonio*. pp. 165–168.

Kin, Kenrick, Björn Hartmann, Tony DeRose, Maneesh Agrawala (2012). Proton: Multitouch Gestures as Regular Expressions. *CHI'12: Proc. SIGCHI Conf. Hum. Factors Comput. Syst. Austin, Texas, USA*. New York: ACM Press pp. 2885–2894.

Krueger, Myron W, Thomas Gionfriddo, Katrin Hinrichsen (1985). VIDEOPLACE An artificial reality. *CHI'85*. New York: ACM Press pp. 35–40.

Mackenzie, Russell, Kirstie Hawkey, Kellogg S Booth, et al. (2012). LACOME: a Multi-User Collaboration System for Shared Large Displays. CSCW'12, Washington. New York: ACM Press, pp. 267–268.

Marr, Stefan, Thierry Renaux, Lode Hoste, Wolfgang De Meuter (2014). Parallel gesture recognition with soft real-time guarantees. *Science of Computer Programming*, vol. 98, no. 2, pp. 159–183.

Marshall, Paul, Yvonne Rogers, Eva Hornecker (2007). Are Tangible Interfaces Really Any Better Than Other Kinds of Interfaces? *CHI'07 workshop on Tangible User Interfaces in Context & Theory, 28 April 2007, San Jose, California, USA*.

Rogers, Yvonne, Youn-kyung Lim, William Hazlewood, Paul Marshall (2009). Equal Opportunities: Do Shareable Interfaces Promote More Group Participation Than Single User Displays? *Human-Computer Interaction*, vol. 24, no. 1, pp. 79–116.

Scheifler, Robert W, Jim Gettys (1990). The X window system. *Software: Practice and Experience*, vol. 20, no. S2, pp. S5–S34.

Schmidt, Kjeld, Liam Bannon (1992). Taking CSCW seriously. *Computer Supported Cooperative Work (CSCW)*, vol. 1, no. 1–2, pp. 7–40.

Scholliers, Christophe, Lode Hoste, Beat Signer, Wolfgang De Meuter (2011). Midas: a declarative multi-touch interaction framework. *TEI'11: Proc. fifth Int. Conf. Tangible, Embed. embodied Interact. Funchal, Portugal*. New York: ACM Press, pp. 49–56.

Scott, Stacey D, Karen D Grant, Regan L Mandryk (2003). System guidelines for co-located, collaborative work on a tabletop display. *ECSCW 2003: Proc. Eighth Eur. Conf. Comput. Support. Coop. Work. Helsinki, Finland*. Springer, pp. 159–178.

Shaer, Orit, Eva Hornecker (2010). Tangible User Interfaces: Past, Present, and Future Directions. *Foundations and Trends in Human-Computer Interaction*, vol. 3, no. 1–2, pp. 1–137.

Sharlin, Ehud, Benjamin Watson, Yoshifumi Kitamura, et al. (2004). On tangible user interfaces, humans and spatiality. *Personal and Ubiquitous Computing*, vol. 8, no. 5, pp. 338–346.

Stanton, Danae, Tony Pridmore, Victor Bayon, et al. (2001). Classroom collaboration in the design of tangible interfaces for storytelling. *CHI'01: Proc. SIGCHI Conf. Hum. factors Comput. Syst. Seattle, Washington, USA*. New York: ACM Press, pp. 482–489.

Strauss, Anselm (1985). Work and the Division of Labor. *The Sociological Quarterly*, vol. 26, no. 1, pp. 1–19.

Tuddenham, Philip, Ian Davies, Peter Robinson (2009). WebSurface. *ITS'09: Proc. ACM Int. Conf. Interact. Tabletops Surfaces. Banff, Alberta, Canada*. New York: ACM Press, pp. 181–188.

Verma, Himanshu, Flaviu Roman, Silvia Magrelli, et al. (2013). Complementarity of input devices to achieve knowledge sharing in meetings. *CSCW'13: Proc. 2013 Conf. Comput. Support. Coop. Work. San Antonio, Texas, USA*. ACM, pp. 701–703.

Vernier, Frédéric, Neal Lesh, Chia Shen (2002). Visualization techniques for circular tabletop interfaces. *AVI'02: Proc. Work. Conf. Adv. Vis. Interfaces Trento, Italy*. New York: ACM Press, pp. 257–266.

West, Joel, Michael Mace (2010). Browsing as the killer app: Explaining the rapid success of Apple's iPhone. *Telecommunications Policy*, vol. 34, no. 5–6, pp. 270–286.

Wobbrock, Jacob O, Andrew D Wilson, Yang Li (2007). Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *UIST'07: Proc. 20th Annu. ACM Symp. User interface Softw. Technol. Newport, Rhode Island, USA*. New York: ACM Press, pp. 159–168.

Xambó, Anna, Eva Hornecker, Paul Marshall, et al. (2013). Let's jam the reactable. *ACM Transactions on Computer-Human Interaction*, vol. 20, no. 6, pp. 1–34.