VIGO: Instrumental Interaction in Multi-Surface Environments

Clemens Nylandsted Klokmose
Department of Computer Science,
University of Aarhus
Åbogade 34, DK-8200 Århus N, Denmark
clemens@cs.au.dk

Michel Beaudouin-Lafon
LRI (Univ. Paris-Sud & CNRS), INRIA
Bâtiment 490
F-91405 Orsay, France
mbl@lri.fr

ABSTRACT
This paper addresses interaction in multi-surface environments and question whether the current application-centric approaches to user interfaces is adequate in this context and present an alternative approach based on instrumental interaction. The paper presents the VIGO (Views, Instruments, Governors and Objects) architecture and describes a prototype implementation. It then illustrates how to apply VIGO to support distributed interaction. Finally it demonstrates how a classical Ubicomp interaction technique, Pick-and-Drop, can be easily implemented using VIGO.

Author Keywords
Ubiquitous Computing, Instrumental Interaction, Multi-surface interaction, UI Architecture, Interaction Paradigm

ACM Classification Keywords
H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

INTRODUCTION

In his seminal paper on ubiquitous computing [28], Mark Weiser envisioned how computers would take on multiple sizes, from the small tab to the notebook-sized pad and the large interactive wall. These devices would be used interchangeably and in combination: pads as "sheets" of paper and tabs as, e.g., input devices for an interactive wall. Part of this vision has been realized today: we now have myriads of small devices, similar in size to the envisioned tabs and pads, and interactive walls are on their way to becoming consumer level products. But the seamless interplay between the multiple device surfaces that Weiser imagined is still far from reality: the simple act of exchanging data among devices typically requires complex configuration or the use of a physical USB key rather than Rekimoto's simple and intuitive *pick-and-drop* [25]. Other examples of multi-surface interactions include the use of a PDA as a personal and portable tool palette [26] or as a remote control [20] when interacting with an interactive whiteboard. While such techniques have been prototyped in the lab, they are still not available to the users at large. We believe this is due to the lack of adequate software support to develop such interactions.

In this paper we specifically address *multi-surface interaction*, i.e. interaction spanning the surfaces of multiple devices. We question the adequacy of the current predominant user interface paradigm, the application-based WIMP interaction model, and its underlying architectural models such as MVC [24] for building user interfaces going beyond a single desktop computer. We examine the requirements for a user interface software architecture that supports multi-surface interaction. We argue that instrumental interaction [4] provides an appropriate framework for interaction in multi-surface environments and introduce *ubiquitous instrumental interaction*. We then present VIGO (Views, Instruments, Governors and Objects), an architecture that supports ubiquitous instrumental interaction and show how it is used to create a generalized version of pick-and-drop.

UBIQUITOUS INSTRUMENTAL INTERACTION

The vision of ubiquitous computing includes the idea that multiple users can interact with multiple devices through a variety of interfaces, including interfaces spanning multiple surfaces. This vision challenges the traditional assumption of one surface / one interface / one application that is very deeply engrained in today's desktop computing environments. This assumption is also pervasive in the software tools used today for creating interfaces – tools which are tightly bound to the platform hosting them and to the WIMP interaction style. These tools typically do not support the multiplicity, dynamism, heterogeneity and distribution that characterize the ideal of multi-surface interfaces, making it difficult in particular to develop multi-surface interfaces.

Two central goals in creating user interfaces for multi-surface environments are to provide users with distributed interfaces that support fluid interaction across stationary and mobile devices and the ability to dynamically configure those interfaces according to the available devices and users' needs. Two major challenges in this context are: Supporting reuse and the quality of learning across different devices [8] (User perspective), and technically supporting the continuity and distribution of work across multiple devices (Developer perspective). We argue that one approach to address these

CHI 2009, April 4 - 9, 2009, Boston, USA.

problems is to *deconstruct* applications and distribute the interface across multiple surfaces rather than simply create scaled-down versions of PC applications to run on, e.g., Personal Digital Assistants (PDAs).

Beaudouin-Lafon [4] introduced *instrumental interaction* to model WIMP and post-WIMP interfaces on desktop computers. The key idea is a conceptual separation between instruments and domain objects. Instruments consist of a physical part (the input devices) and a logical part (the software component). Instruments are *mediators* [7] between the user and domain objects: The user acts on the instrument, which transforms the user's actions into commands that affect the relevant domain objects and provides feedback to the instrument and the user.

Instrumental interaction was inspired by the way we use physical tools: A painter can freely add or remove brushes from his collection, pass them around, etc.; Brushes are not bound to painting on a canvas, they can also be used to paint on the wall or on the hand of the painter. Computer applications do not currently support this level of flexibility: a brush in a drawing application can rarely be removed and used in another context, such as a text editor. Applications typically have a predefined set of tools that is difficult or impossible to adapt to one's needs or taste. This lack of flexibility limits the mobility, distribution and customizability of interfaces. It also typically results in complex monolithic applications, built for general-purpose personal computers, with dozens or even hundreds of tools to cover all possible needs.

While instrumental interaction was developed in the context of desktop interfaces, the concepts are more general and apply particularly well to multi-surface interaction. We call *Ubiquitous Instrumental Interaction* our extension of instrumental interaction to this context. In Ubiquitous Instrumental Interaction, the instrument is an explicit construct rather than a descriptive concept. Ubiquitous instruments should be exchangeable among users, they should work in a similar way on different devices, sometimes across multiple devices, and they should be technically separated from the objects they operate on. Instruments should be applicable to domain objects and on device surfaces when and where it makes sense, even if they were not designed to do so in the first place. The objects that users interact with through instruments should be able to migrate across device surfaces, support multiple views, and be manipulatable by an instrument in ways not necessarily anticipated by the object. Note that this does not preclude an instrument from "breaking" an object, i.e. to make it unusable by other instruments[1]. Finally, instruments should themselves be objects and therefore be manipulatable by other instruments.

While we recognize the need for specific instruments that work only with certain objects on certain surfaces for performing specialized operations, we also want to support the ability for instruments to be more open-ended and usable in ways not anticipated by their designers. For example, pick-and-drop is a very flexible and generic instrument capable of picking up an object of any type on any of the surfaces available to the user and dropping it onto another object on another surface. A color picker is another example of a generic instrument that can be used in many contexts. It typically works with a color palette that displays the set of available colors, but can also be generalized to pick the color of any object with a color attribute. A last example is an annotation instrument that can add annotations to any object, e.g. by supporting hand drawing on any surface or adding electronic Post-it Notes to any object. Such flexibility supports what Illich [14] calls *convivial use*: "*Tools foster conviviality to the extent to which they can easily be used, by anybody, as often or as seldom as desired, for the accomplishment of a purpose chosen by the user. The use of such tools by one person does not restrain another from using them equally.*"

Architectural Requirements

Implementing Ubiquitous Instrumental Interaction requires a software architecture that enables the flexibility that we have described above in the context of a distributed infrastructure. We believe that a software architecture based on small-grain components that can be reconfigured according to the users' needs or the available devices is the most appropriate. We identify two main requirements for this architecture:

Decoupling: Objects and instruments should be separate components that communicate through a simple protocol allowing instruments to query and modify objects. For example, any object that provides a 2D surface can be used by a pen instrument to add annotations. Such decoupling will facilitate the distribution and replication of objects, or parts of objects, across multiple devices.

Integration: Despite the fact that interaction may involve multiple surfaces, multiple processes and multiple machines, the system should appear as a single consistent entity from the user perspective. The ability to use the same instrument with objects of different types should be seamless and in general the system should support a seamless user experience.

RELATED WORK

While most research on multi-device interaction has focused on *migrating* applications across devices, especially through model-based approaches [18], little research has addressed true multi-surface interaction, i.e. interactions that involve multiple surfaces. Notable exceptions include the Pebbles project [20] and Demeure et al.'s work on distributed user interfaces [9].

Some systems have attempted to provide a generic solution to interact with applications in a multi-device computing environment. XWeb [23] and The SpeakEasy Recombinant Computing Framework [21] are probably the closest to our work in that respect. XWeb decouples internet services from their user interface to facilitate access from multiple kinds of devices by introducing device-independent widgets that specify the possible values of a service's data items, e.g., a date

[1]This may be useful, for example in the board game that we describe later, where a player may want to create art with the pieces rather than play the game.

or time. In SpeakEasy services can provide their own user interfaces to be aggregated on the client. The goal of XWeb and SpeakEasy however is to automatically transform user interfaces for different devices, while we focus instead on a uniform interaction model to create custom interfaces that leverage the capabilities of the environment.

Other systems, such as the iStuff toolkit [2], are designed to explore multi-device interaction but do not embody a specific interaction model. Cameleon-RT [3] on the other hand is a reference model rather than an implementation framework. It focuses on the automatic adaptation of plastic interfaces while we focus on adaptability by the users.

More generally, architectures for Ubicomp systems have focused mainly on middleware to support system requirements such as distribution, discovery, fault-tolerance or context-awareness, but do not address the specific needs of interaction (see, for example, [17]). The BEACH architecture [27] is a rare exception as it addresses interaction explicitly although it seems limited to classical interaction techniques based on mouse and gesture input.

Architecture models for user interfaces have a long history, with the MVC (Model-View-Controller) design pattern [15] being by far the most widespread solution. A well-known problem with MVC is the strong dependency between the view and controller that limits reusability. Abstraction-Link-View (ALV) [13] was designed for sharing a common model (the abstraction) among multiple networked clients potentially each with their own view, but is otherwise quite similar to MVC. None of the existing patterns however make instruments explicit, instead they promote a widget-based type of interaction. Finally Document-Presentation-Instruments (DPI) [6] is a document-centric software architecture based on instrumental interaction. Like our approach the goal it decouples instruments from the target objects, however DPI is a desktop-based framework and does not address the distribution of objects and instruments across multiple machines.

THE *VIGO* ARCHITECTURE

We now present *VIGO* (Views, Instruments, Governors and Objects), the architecture that we have designed to implement Ubiquitous Instrumental Interaction. VIGO is an alternative to MVC designed to create distributed interfaces based on the principles described in the previous section.

Figure 1 presents the VIGO architecture. *Objects* are presented to the user through *views*. Users manipulate objects through *instruments*, which query views to identify the objects being designated. In order to manipulate an object, an instrument queries the *governors* attached to that object to validate its manipulations. Governors, on the other hand, observe object changes to implement potential side effects. Finally governors can manipulate their attached objects if the user's actions on the object have side effects beyond that object. The following description shows that this design ensures a strong separation of concerns, provides great flexibility and supports distribution among multiple devices and machines.

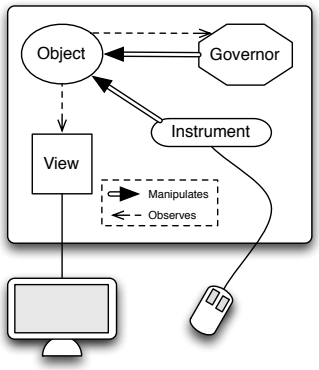

Figure 1. VIGO in pseudo-UML

Objects

VIGO objects are different from objects in the classical object-oriented sense. In order to achieve the required decoupling between objects and the instruments manipulating them, objects are passive, i.e. they do not provide operations or methods applicable to them. Rather than hiding their state through encapsulation, they expose it as a set of directly accessible properties, while the behavior that is usually provided through methods is encapsulated in *governors* (see below). Compared with MVC, the Model is separated in VIGO into Objects, implementing state, and Governors, implementing behavior.

Objects can be primitive, consisting only of properties, or composite, consisting of properties and other objects. An object can be a part of several objects simultaneously, such as when a diagram is used in multiple documents: a change in the diagram is then reflected in all the parent documents. This is a different concept than multiple views, which is discussed below. This object structure lends itself to a natural description in XML, which is the format we use for persistence. Examples of concrete objects are text documents, graphic canvas, the board and pieces of a board-game (which we will see later) or concrete user interface elements.

Objects (or parts of objects) can be distributed over multiple computers. In our implementation their state is kept synchronized across the replicas. Because objects are passive, this can be implemented simply and efficiently.

The underlying principle behind our notion of object is that every interaction must target one or several objects and may result in a change to the state of these objects. Rather than interacting with a "system" or "application" as with traditional interfaces, the user interacts with objects, by means of instruments. The fact that objects are open gives tremendous power to instruments. On the one hand, it makes it possible to "break" an object by putting it into an inconsistent state. Governors are designed to control this potential chaos. On the other hand, it makes it possible to implement interactions that were not anticipated. For example, in a board game, the board could be used to play a different game (the rules will be embodied, as we will see, in governors), or it may be

possible to annotate the board with arbitrary marks. The traditional approach where objects are only accessed through methods clearly precludes such unanticipated uses.

Views

Views on objects are translations of the objects into one or more modalities perceptible by the user. The are very similar to views in MVC. A typical instance of a view is a visualisation of an object (and its subobjects) on a screen. Views are strongly coupled with the objects they represent, so that any change to the object is reflected in the view. Views on the other hand do not provide any kind of interaction: any change to the view is the result of a change to the object. Views provide a service to translate coordinates (in the case of a visual display) between the view of and object and that of its subobjects or vice-versa.

Unlike objects, views are device-dependent, i.e. they optimize their representation of the object for the display device at hand. Multiple views can be associated with an object, in which case they are synchronized. Note however that multiple views of the same object can also occur when an object is shared among several parents, as described in the previous section. Since objects are passive and views are pure representations, they can be implemented efficiently, e.g. through an observer pattern [11]. On the other hand, for efficiency reason, the object should be available on the machine that holds the view. This is easily achieved using the ability to distribute an object, as described in the previous section.

Instruments

Beaudouin-Lafon [4] defines an instrument as:

> *... a mediator or two-way transducer between the user and domain objects. The user acts on the instrument, which transforms the user's actions into commands affecting relevant target domain objects.*

The concept of instrument is inspired by the real world: a stick to enhance one's reach, a pen to write on a piece of paper, a hammer to drive a nail. Examples of digital instruments include those to enter text, manipulate graphics, draw, select objects, etc. The tools in the tool-palette in applications such as Photoshop are typical examples of instruments.

An instrument may need to provide feedback to the user by presenting some information. This is achieved by associating instruments with objects and creating views for these objects. This means, in addition, that an instrument can manipulate another instrument through the object associated with it and the proper governors.

One way of concretely thinking of instruments is as event processors. Instruments react to input from the user or from other instruments, change objects and fire new events for other instruments to react to. Interaction occurs through chains of instruments, e.g. an instrument for selecting an object on the screen is chained with an instrument for moving objects.

An instrument can be distributed, i.e. it may require input or output from/to multiple devices connected to different machines and yet function as a single instrument. For example, a PDA used as a remote control will require some feedback on the controlled device, while the pick-and-drop technique [25] uses input from two devices, the source and destination. Since instruments are event processors, this requires that the event system must be distributed, i.e. that events produced by a source on one machine are transmitted to a destination on a different machine. Since events are asynchronous, this can work with any network transport system.

Instruments have no direct equivalent in MVC. Instruments that correspond to traditional widgets, such as a scrollbar, can be implemented with an MVC triplet where the controller is the instrument itself while the model and the view implement its associated object. But MVC forces such an instrument to be linked to a target object, such as a text area, through a parent MVC controller. This is one of the reasons why a simple interaction such as drag-and-drop is difficult to implement with MVC (in fact, it does not really fit the pattern). By contrast VIGO instruments are loosely coupled with their targets.

In other cases, the equivalent of instruments are implemented in the controller of an object in MVC. For example, an implementation of a text area with MVC typically include the text editing commands in the controller. This merge of a domain object (the text itself) and the instruments to manipulate it (the text editing commands) in a single MVC entity does not provide the separation of concerns that VIGO encourages. It also limits extensibility, e.g. adding a highlighter tool can be easily accomplished with VIGO with an independent instrument while it requires changing the code of an existing controller in MVC.

Governors

We have now described how objects are passive constructs visualized through views and manipulated by the user with instruments. The manipulations issued by an instrument consist in changing the state of the targeted object. Specific rules governing these state changes or the consequences of these state changes are not the responsibility of the instrument, otherwise it would be very difficult to create polymorphic instruments [5] that are independent from the objects they manipulate. Consider for example a board game such as checkers or Othello. If the instrument used to move the pieces of the game implements the rules of the game, then it cannot be used for other games, or indeed for anything else. Another solution is to implement the rules in the board object, but this breaks our object model.

To solve this problem we introduce *governors*[2] to embody the rules and consequences of interactions with objects. Governors implement the "application logic" or "business rules" commonly found in the MVC Model.

[2]The name *governor* was chosen since this construct adds the cybernetic reactive aspects of the system. The word cybernetics stem from the Greek *kybernetes* meaning steersman, *governor*, pilot, or rudder [16]

Governors are associated to objects at the level of individual properties: a client, typically an instrument, that needs access to a property of an object asks the governors associated with that property whether that change is acceptable and what an acceptable change would be. Once the change is made it notifies the governors so they can take any additional action. The idea is that a governor controls certain aspects of an object and only the properties that are relevant to that governor are associated with it. Several governors may be associated with one object and several objects may be associated with one governor. Finally governors are stateless, i.e. all state that governors need to manage must be stored in a separate object or added to an existing object.

Let us illustrate this with the Othello[3] game. The positions of the pieces on the game board are associated with a governor handling the game rules and the consequences of manipulating the pieces. The move instrument queries the piece's governor when it is about to drop the piece on the board. The governor checks that it is a legal move and if so returns the proper position for the piece, i.e. the middle of the destination square. If it is not a legal move, it returns the list of valid moves, which the instrument may decide to highlight. Once the piece has been set to the new position, the governor is again notified of the change, and it applies the side effect, i.e. turns over the pieces according to the rules. Note that since it is up to the instrument to query the governor, it could decide to bypass it and cheat or even break the game. Note also that the same move instrument used to move the pieces can be used to move the whole board. One could imagine a governor for the board that "shakes" the pieces when the board is moved, as often happens with a real board. Finally if the piece governor is to keep track of turn-taking in the game, it must store this state in a separate object or in the board itself. This ensures that if another instrument manipulates the board, it can access that state as well.

Governors should not be seen as direct mediators between instruments and objects: the governors are not an interface to the objects, neither are they transparent to the instrument and just react to the manipulations of the objects. Both cases would lead to a lack of flexibility. In the first case, instruments would not be able to bypass the governors; in the second case instruments would not be able to visualize what the governor proposes, such as the valid moves in the Othello game. Instead, governors and instruments negotiate: instruments query the associated governors to validate a manipulation or to get the valid or suggested manipulations (valid moves in the case of the Othello game) which they could in principle ignore (which in Othello would break the game).

Since governors are stateless there are various ways to handle governed distributed objects. Each machine could have an instance of a shared object's governors, only one of the machines could hold the governors and let the others query it, or the governors could reside on a central server. Note that governors can be attached and detached from objects dynamically. In the case of the Othello game, detaching the pieces' governor allows to use the move instrument to move the pieces freely, while attaching another governor would allow to play a different game.

[3] Also known as *Reversi*.

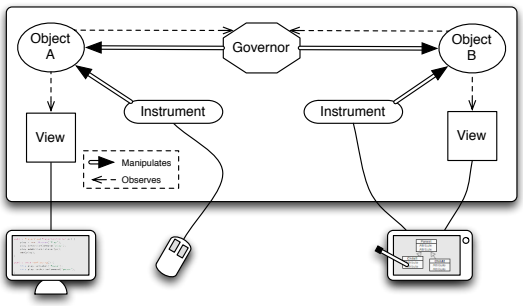

Figure 2. Two objects representing each other through a governor

Using governors to synchronize objects

Multiple representations of a single object are common in computer applications. For example, a UML specification may be represented as a text or as a diagram, a chess game may be represented by an animated board or a list of moves, etc. Representations are also often used to create computer renderings of physical objects, such as the reading of a sensor. Such representations involve a fairly large semantic distance between the object being represented and the representation, to the point where, in the user's mind, the different representations and the original object are separate (but related) objects. In VIGO, this notion of representation does *not* correspond to the notion of view. Since instruments manipulate objects through their views, the mapping between a view and its object must be fairly straightforward.

In order to support the kind of representation that involves a large semantic distance, we use objects and governors instead. Consider the case of a mapping between object-oriented code and an interactive UML diagram: The code and diagram are edited through text-editing instruments and diagram instruments respectively. The code has a text view and associated governors for handling syntax highlighting, indentation, etc. while the UML diagram has a graphical view and governors for aligning boxes and snapping edges to boxes. The two objects, however, share a governor handling the mapping between code and UML (Figure 2) so that adding a new box in the UML class diagram creates the associated code and visa versa. Controlling physical devices is analogous: The external device is represented by an object with an associated governor handling the synchronization of state between object and machine.

USING VIGO: REALIZING PICK-AND-DROP

In this section we describe our prototype implementation of the VIGO architecture. Our hardware setup consists of a SMART Board™ connected to a Mac together with a Nokia N810 Internet Tablet. The implementation is a client-server system developed in Python using the Twisted [10] distributed computing framework. For visualization we use the Apple Cocoa framework on the Mac and PyGTK + Cairo on the Nokia N810. The server has three facets, an object-server, an event-server and a governor-server.

Objects

Objects are defined in a simple XML language with primitives such as basic shapes and text. Using graphical objects simplifies the mapping to views and is sufficient for our experiments. Graphical objects are laid out in a Canvas, which is itself contained in a view-object. View-objects are the only objects that are not replicated, they are created locally on a device when a canvas is loaded.

Objects have a unique id and are replicated across clients by the object-server. Each client manipulates a local object and the changes are propagated to other clients sharing the same object through the object-server.

Views

Views are device-specific components that display objects and provide methods for translating between screen and view coordinates and. In our implementation, they also implement *picking*[4]. Our implementation is naive in that it redraws the whole view when the corresponding object changes. This could be improved by observing the changes in the object and optimizing redrawing. Multiple views on a single object are supported. Since views are the only device-dependent construct and have a fairly small interface, it is easy to port VIGO to another platform using a different graphical toolkit. This is what we have done with Cocoa on one device and PyGTK and Cairo on another.

Instruments

Instruments take input events such as button presses and transform them into object manipulations. This transformation is described by a state machine. We have implemented a Python library for state machines similar in its principle to SwingStates [1]. We have noticed that state machines not only reduce the traditional "spaghetti of callbacks" problem of user interfaces [19], they also provide a good hint of the complexity of the instrument being implemented.

An example of a simple instrument is an instrument for moving objects on the screen. This instrument actually consists of two instruments: An instrument to select an object on the screen, and an instrument to move the selected object. The selection instrument processes input events and fires a <select> event when something has been selected and can now be moved. The move instrument is triggered by the <select> event and tracks the mouse until the button is released (Figure 3). From the user's perspective, these two instruments act as a single, integrated one. Separating them has the advantage that they can be reused more easily to create more complex instruments.

Instruments can share events through the event-server hence they can receive local as well as remote events. The latter are sent by other clients and automatically dispatched by the event server.

[4]Picking could be implemented in a dedicated instrument, however it is more efficient to take advantage of the point-shape intersection computation of the underlying graphical toolkit. Advanced selection, e.g. of hidden objects, and selection based on queries are implemented in instruments

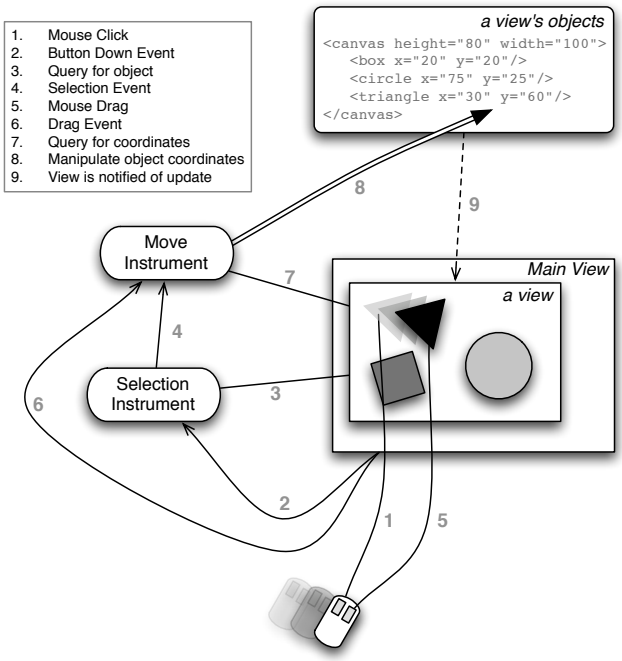

Figure 3. Ungoverned interaction between move instrument and object

We can now show the implementation of the pick-and-drop instrument. We use two state-machines (Figure 4). Selecting an object when in the $picking$ state triggers the transition to the $dropping$ state and a $Pick$ event is fired with the *id* of the picked object. The remote pick-and-drop instrument receives this $Pick$ event and transitions to the $picked$ state. The picked object can now be dropped with the remote pick-and-drop instrument by loading the object whose *id* is in the $Pick$ event from the object-server. When one of the state machines drops the object, it fires a $Drop$ event that reverts all instruments to the $Start$ state. Figure 5 gives the Python pseudo-code for the pick-and-drop state machine. It uses our syntax for defining states and transitions in instruments with Python decorators ($@state$ declares a state while $@transition$ declares a transition).

The color-picker instrument is implemented in a similar way. On the N810 PDA we have implemented an instrument that fires events indicating that a color has been picked, and on the SmartBoard a drawing instrument that reacts to these events and changes its color (Figure 10). The color-picker can work on anything that has a color attribute, hence a specific palette object is not required. We have implemented other instruments for resizing shapes, terminating views and activating other instruments. The latter uses gesture input on the SmartBoard whereas on the PDA it uses the keyboard.

Instruments can provide visual feedback by creating an associated object and using the same vector-graphics language as other objects. The view for this object is a transparent overlay above the other views. This is used, e.g., by the gesture instrument to provide feedback about the gesture being made and whether the gesture has been recognized or not.

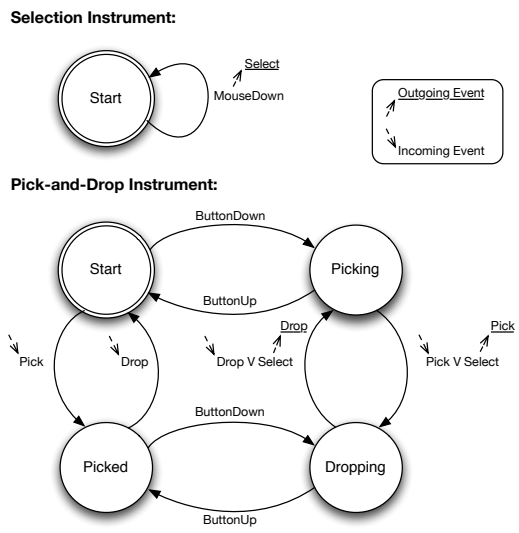

Figure 4. Selection and pick-and-drop state machines

Governors

Governors provide a simple interface for instruments to validate changes and to retrieve sets of valid changes. In the Othello example a move instrument can ask the governor of a piece whether a position is valid and can query for the set of currently valid positions. When a change to a governed object has been made, the governors are notified and can react accordingly, e.g. to change the color of one or more pieces on the Othello board.

The Othello game uses two governors: The board-governor makes sure that pieces can only be placed within the squares of the board, and adds a new property to the pieces on the board to hold their grid coordinates ($A1$-$H8$). It also places a fresh piece in the corner of the board when one has been moved to a cell. The second governor is attached to the pieces and implements the Othello rules: checking that a move is correct and turning over the proper pieces after a piece has been put down. It uses the grid coordinate properties added to the pieces by the board governor. Figure 6 illustrates three boards, the first board has no governors (pieces can be placed anywhere), the second board only has the board-governor (pieces are in the middle of the squares) and the third board has both governors (it shows a correct game). Figure 7 shows an XML object attached to both governors.

The interaction between the move instrument and governors is as follows: The move instrument queries the governor associated with the x-y position of the piece being moved for valid values. The board governor computes the mapping from the grid coordinates stored in the piece ($A1$-$H8$) to the x-y position on the board. It then asks the governors associated with the grid coordinate properties whether their position is a valid move. It collects, say, the list ($C5$, $H1$). The board governor now takes the intersection between its valid grid coordinates and those returned by the governors and returns their translation into x-y positions. The instrument can now use the returned positions as feedback to the user or to

```
@state    # defining state "start"
def start(self):
  # transition on button press to state "picking"
  @transition(event=ButtonDown, to=self.picking)
  def action(event)
    pass  # no action

  @transition(event=Pick, to=self.picking)
  def action(event)
    self.picked = event.picked # record picked object

@state    # defining state "picking"
def picking(self):
  @transition(event=Select, guard=self.picktest, to=self.dropping)
  def action(event):
    self.picked = objectHandler.getObj(event.element)
    eventHandler.fireEvent(Pick(self.picked)) # fire Pick event

  @transition(event=Pick, to=self.dropping)
  def action(event):
    self.picked = event.picked

  @transition(event=ButtonUp, to=self.start)
  def action(event):
    pass

@state    # defining state "dropping"
def dropping(self):
  @transition(event=Select, guard=self.picktest, to=self.picking)
  def action(event):
    if self.picked is not None:
      <insert picked object into selected object>
      eventHandler.fireEvent(Drop(self.picked)) # fire Drop event

  @transition(event=Drop, to=self.picking)
  def action(event):
    pass

@state    # defining state "picked"
def picked(self):
  @transition(event=ButtonDown, to=self.dropping)
  def action(event):
    pass

  @transition(event=Drop, to=self.start)
    pass
```

Figure 5. Pseudo-code for pick-and-drop instrument

place a piece. When placing the piece, the board-governor is notified, sets the new grid coordinates for the piece and notifies the Othello governor which exchanges the pieces on the board according to the game rules. This shows that the board governor can be used not only for playing Othello, but also for any game that relies on a board with a square grid.

Instruments can query governors for valid values in order to provide feedback to the user about the rules associated with an object. In the Othello game this is used by the move instrument to show whether a position is legal: in Figure 6, the move instrument turns the line red if the user points at an illegal position.

Instruments interact with governors through a client-side governor-handler (Figure 8). In our implementation governors reside on a governor-server, and are instantiated when a client loads an object for the first time. Governors could ho-

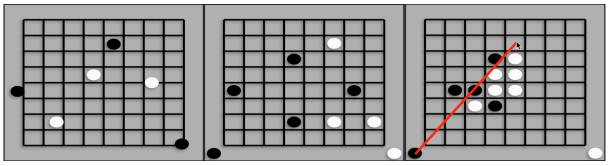

Figure 6. Othello Governors

```
<circle x="−10" y="−10" radius="7" boardPlacement="D4" fillColor="
  #000000" id="6bd0">
  <governed by="boardGovernor" instance="d27d">
    <governs attribute="x"/>
    <governs attribute="y"/>
  </governed>
  <governed by="othelloGovernor" instance="d09e">
    <governs attribute="boardPlacement"/>
  </governed>
</circle>
```

Figure 7. Object with governors

wever be cached locally, e.g. in case the network connection is temporarily lost. If multiple governors are associated with a set of properties the governor-handler computes the intersection of the valid values that they return. If an instrument wants to change the properties of an object that is not governed, e.g. drawing annotations on the board of the Othello game, it can do so without having to negotiate with the governors. An instrument can also simply ignore the governors and change object properties directly.

Supporting multi-surface interaction

Now that we have described the components, we can illustrate some multi-surface interactions supported by the architecture. We start with the pick-and-drop instrument. Since any object can be used with that instrument, we can apply it to the Othello board itself. This works with no extra code and allows to instantly share live applications across multiple devices and surfaces: picking an ongoing game on the SmartBoard and dropping it on the PDA, we instantly have a networked multiplayer version of Othello (Figure 9).

Another application is to put the pieces of the Othello game on the PDA of each player. They just use pick-and-drop to move the pieces from their PDA to the SmartBoard, again without modifying anything to the existing game. We have also implemented a printer object associated to a print gover-

```
# Retrieve the governors associated with the object's x and y properties
governors = governorHandler.getGovernors(obj, ["x", "y"])
...
# The instrument checks whether the new object position is valid
if governors.validVal(["x", "y"], x, y):
  object.set("x", x)
  object.set("y", y)
  # It notifies the governors that the object has changed
  governors.changed(obj)
```

Figure 8. Pseudo-code illustrating the interaction with governors

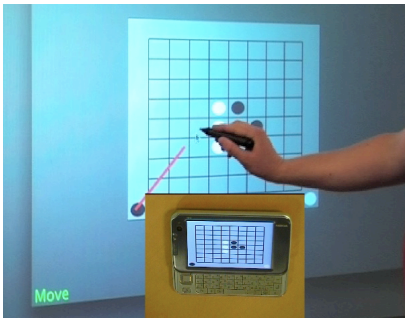

Figure 9. Shared game between the SmartBoard and the N810 PDA

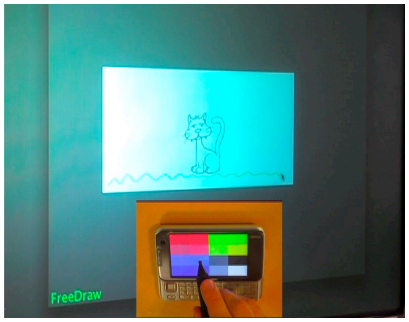

Figure 10. Image created on the board using palettes on the PDA

nor. The print governor simply prints the object that is dropped on the printer. The printer can be picked-and-dropped like any other object and therefore can easily be shared across multiple devices and taken away on one's PDA.

Using these components, we have created a scenario where the PDA is used as both a color and object palette for creating graphics on the SmartBoard. Images and shapes are picked on the PDA and dropped on the SmartBoard, while colors for freehand drawing are selected on the board (Figure 10). Finally the canvas on the SmartBoard can be picked and dropped on the PDA, or dropped on the printer for printing.

DISCUSSION AND QUALITATIVE EVALUATION

VIGO was designed specifically to address multi-surface interaction and support a flexible interaction style where users can use any instrument on any object. We first compare it with MVC and then provide some formative evaluation.

VIGO is conceptually close to MVC but has a number of key differences. First, VIGO was designed from the ground up for distributed interfaces. While there are some implementations of distributed MVC [12], in particular in the context of Rich Internet Applications, they usually have a number of limitations with respect to the original pattern, and to our knowledge none of them natively support multi-surface interactions such as pick-and-drop. Second, VIGO stresses the notion of mediation that is absent in MVC. In MVC the View is both a visualization of the model and of the "tool" to interact with the model, whereas VIGO puts the visualization of the model in the View and the visualization of the tool in the Instrument. This makes it possible for multi-

ple, independent instruments to interact with the same model through the same view, which is different from MVC's model-sharing through multiple views. MVC makes it difficult, or at least non-natural, to create non widget-like interactions. For example, drag-and-drop types of interaction are not well supported by MVC, but they are a key interaction in multi-surface environments. Finally, the MVC state-based Model is not explicitly present in VIGO. Instead, the Governor represents behavioral aspects of the model that are relevant to interaction. This provides a high degree of flexibility that is difficult to obtain with MVC.

We now summarize a qualitative evaluation of VIGO based on some of the criteria defined by Olsen [22] to assess user interface systems research:

Generality of the solution: Instrumental interaction makes the assumption that interaction is mediated by an instrument. Whether all interaction with a computer is mediated or not is beyond the scope of this paper[5], however it does cover a wide range of interaction styles, including traditional WIMP interaction, tangible interaction and pen-based interaction [4] that are relevant to multi-surface environments. Our experience so far is that VIGO provides appropriate support to implement ubiquitous instrumental interaction, and we have yet to find an instrumental technique that does not fit the pattern.

To give a perspective on how ubiquitous instrumental interaction could change some basic uses of the computers, consider instant messaging: with VIGO, a conversation would be an object that two or more users can share (a generic instrument would support a directory and a means to share objects). The governors associated with the conversation would add a user name and time stamp to the entries added to it. Entries could be images, text, drawings or even a live game like Othello, added with pick-and-drop. The conversation itself could be moved around between the surfaces available to the user and manipulated with the instruments available on the devices at hand, e.g. freehand writing on a PDA and keyboard text entry on a laptop.

Viscosity: According to Olsen viscosity includes flexibility, expressive leverage and expressive match. The goal of VIGO is to clearly separate concerns among its strongly decoupled components. We have illustrated the level of flexibility this provides. In particular, adding or removing governors radically changes the behavior of objects, and instruments can operate on objects they know very little about. New instruments can easily be implemented and tested with existing objects, instruments and governors. Expressive leverage is demonstrated by the polymorphic aspect of instruments and the ability to reuse the various components. For example, the draw instrument that was created for the paint application can be used to annotate the Othello board, and the color picker can pick the color of any other object. The flexibility of the architecture made it easy to create a pick-and-drop instrument that can move pieces of a board game as well as the complete live game from one device to another. Expressive match is supported by the use of concepts familiar to developers, such as a (distributed) event system, state machines and XML-like objects, and the "concrete" character of the components that makes it easy to decide what should be in the objects, the governors and the instruments.

Power and Scale: The power of instrumental interaction is the ability to apply any instrument to any object, and the power of VIGO is to provide developers with simple means to achieve this interoperability. Combining objects, combining governors and combining instruments is simple, what may prove more challenging is controlling which interactions are desirable and which are not. While we see evidence that the approach is scalable, we need to validate it by implementing a larger system. In case the flexibility of the model proves a weakness rather than a strength, for example if we loose control over the combinatorial explosion of interactions between instruments, governors and objects, we will consider adding appropriate control mechanisms.

CONCLUSION AND FUTURE WORK

This paper addresses a problem area that has gained little attention despite the development of ubiquitous computing technologies. We have shown the potential of multi-surface interaction and presented an extension of instrumental interaction called Ubiquitous Instrumental interaction that supports distributed interaction among multiple devices and computers. It has then presented VIGO, a software architecture pattern designed for the implementation of ubiquitous instrumental interaction and illustrated it with several examples. VIGO supports reusability of instruments by users, since the same instrument can be used with different objects. It also supports reusability of components by the developer, in particular through the flexibility provided by the dynamic management of governors attached to objects. Finally we have shown how to implement the classical Ubicomp interaction technique pick-and-drop and discussed a number of evaluation criteria for the proposed pattern. We plan to continue the development of VIGO to further explore how to best support multi-surface interaction and address issues such as the configuration of instruments, the scalability of the architecture and the design of novel multi-surface interaction techniques.

ACKNOWLEDGMENTS

We thank Pär-Ola Zander, Susanne Bødker, and Olav Bertelsen for fruitful discussions. We thank Pierre Dragicevic, Rob Jacob, Allan Hansen and Wendy Mackay for insightful inputs on the paper, Rasmus Berlin for work on an initial implementation and Jonas Petersen for AV assistance.

REFERENCES

1. C. Appert and M. Beaudouin-Lafon. Swingstates: adding state machines to the swing toolkit. In *Proc. ACM User Interface Software Technology (UIST'2006)*, 319–322, New York, NY, USA, 2006. ACM.
2. R. Ballagas, M. Ringel, M. Stone, and J. Borchers. istuff: a physical user interface toolkit for ubiquitous

[5]Speech-based interaction, for example, does not seem to be mediated, unless one considers the speech recognizer itself as an instrument

computing environments. In *Proc. ACM conference on Human factors in computing systems (CHI'03)*, 537–544, New York, NY, USA, 2003. ACM.

3. L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary. CAMELEON-RT: a software architecture reference model for distributed, migratable, and plastic user interfaces. In *EUSAI 2004, LNCS 3295*, 291–302, 2004.

4. M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI'2000)*, 446–453. ACM Press, 2000.

5. M. Beaudouin-Lafon and W. E. Mackay. Reification, polymorphism and reuse: three principles for designing visual interfaces. In *Proc. ACM Conference on Advanced Visual Interfaces (AVI'2000)*, 102–109, New York, NY, USA, 2000. ACM.

6. O. Beaudoux and M. Beaudouin-Lafon. Dpi: A conceptual model based on documents and interaction instruments. In *Proc. Computers XV Interaction without frontier (HCI 2001 and IHM 2001)*. Springer Verlag, 2001.

7. S. Bødker. *Through the Interface. A Human Activity Approach to User Interface Design*. Lawrence Erlbaum Associates, Inc., 1991.

8. C. Brodersen, S. Bødker, and C. N. Klokmose. Quality of learning in ubiquitous interaction. In *Proc. European Conference on Cognitive Ergonomics (ECCE)*, 2007.

9. A. Demeure, J. Sottet, G. Calvary, J. Coutaz, V. Ganneau, and J. Vanderdonckt. The 4C reference model for distributed user interfaces. In *Int. Conf. on Autonomic and Autonomous Systems*, 61–69, 2008.

10. A. Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.

11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

12. T. C. N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proc. ACM symposium on User interface software and technology (UIST'1998)*, 1–10, New York, NY, USA, 1996. ACM.

13. R. D. Hill. The abstraction-link-view paradigm: using constraints to connect user interfaces to applications. In *Proc. ACM Human factors in computing systems (CHI'92)*, 335–342, New York, NY, USA, 1992. ACM.

14. I. Illich. *Tools for Conviviality*. Fontana/Collins, 1973.

15. G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

16. Merriam-Webster. *Dictionary and Thesaurus*. 2008. `http://www.merriam-webster.com/`.

17. M. Modahl, B. Agarwalla, G. Abowd, U. Ramachandran, and T. S. Saponas. Toward a standard ubiquitous computing framework. In *Proc. ACM Workshop on Middleware for pervasive and ad-hoc computing (MPAC'2004)*, 135–139, New York, NY, USA, 2004. ACM.

18. G. Mori, F. Paternò, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. on Soft. Eng.*, 30(8):507–520, 2004.

19. B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM Symposium on User interface software and technology (UIST'1991)*, 211–220, New York, NY, USA, 1991. ACM.

20. B. A. Myers, J. Nichols, J. O. Wobbrock, and R. C. Miller. Taking handheld devices to the next level. *Computer*, 37(12):36–43, 2004.

21. M. W. Newman, S. Izadi, W. K. Edwards, J. Z. Sedivy, and T. F. Smith. User interfaces when and where they are needed: an infrastructure for recombinant computing. In *Proc. ACM symposium on User interface software and technology (UIST'2002)*, 171–180, New York, NY, USA, 2002. ACM.

22. D. R. Olsen. Evaluating user interface systems research. In *Proc. ACM Symposium on User interface software and technology (UIST'2007)*, 251–258, New York, NY, USA, 2007. ACM.

23. D. R. Olsen, S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal interaction using xweb. In *Proc. ACM Symposium on User interface software and technology (UIST'2000)*, 191–200, New York, NY, USA, 2000. ACM.

24. T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

25. J. Rekimoto. Pick-and-drop: A direct manipulation technique for multiple computer environments. In *Proc. ACM Symposium on User Interface Software and Technology (UIST'97)*, 31–39, 1997.

26. J. Rekimoto. A multiple device approach for supporting whiteboard-based interactions. In *Proceedings of the 1998 ACM Conference on Human Factors in Computing Systems (CHI'98)*, 1998.

27. P. Tandler. Software infrastructure for ubiquitous computing environments: Supporting synchronous collaboration with heterogeneous devices. In *Proc. International conference on Ubiquitous Computing (UbiComp'2001)*, 96–115, London, UK, 2001. Springer-Verlag.

28. M. Weiser. The computer for the 21^{st} century. *Scientific American*, 265(3):66–75, Feb. 1991.