Shared Substance: Developing Flexible Multi-Surface Applications

Tony Gjerlufsen[1] tony@cs.au.dk
Clemens Klokmose[2,3] clemens@klokmose.net
James Eagan[2,3] eaganj@lri.fr
Clément Pillias[3,2] pillias@lri.fr
Michel Beaudouin-Lafon[2,3] mbl@lri.fr

[1]Univ. Aarhus
DK-8200 Aarhus, Denmark

[2]LRI, Univ. Paris-Sud & CNRS
F-91405 Orsay, France

[3]INRIA
F-91405 Orsay, France

ABSTRACT

This paper presents a novel middleware for developing flexible interactive multi-surface applications. Using a scenario-based approach, we identify the requirements for this type of applications. We then introduce *Substance*, a data-oriented framework that decouples functionality from data, and *Shared Substance*, a middleware implemented in Substance that provides powerful sharing abstractions. We describe our implementation of two applications with *Shared Substance* and discuss the insights gained from these experiments. Our finding is that the combination of a data-oriented programming model with middleware support for sharing data and functionality provides a flexible, robust solution with low viscosity at both design-time and run-time.

Author Keywords

Multi-surface interaction, Data-oriented model, Middleware

ACM Classification Keywords

H.5.2 Information Interfaces & Presentation: Miscellaneous

INTRODUCTION

Multi-surface environments are ubiquitous computing environments where interaction spans multiple input and output devices and can be performed by several users simultaneously. Such environments are becoming more common, not only in specially-equipped rooms but also in day-to-day situations, such as when a small group of people use their laptops and smartphones to share data and work collaboratively.

A key requirement of such multi-surface environments is *flexibility*: data, computation, interaction and visualization are distributed across heterogeneous devices and should be dynamically reconfigurable according to the users' needs, *i. e.* it should be possible to add and remove hardware devices as well as software features at run-time [1, 5]. For example, a user should be able to move a window from a laptop to a large wall display to take advantage of its higher resolution, attach new functionality to this shared content, such as running a particular analysis, and interact with the data on the wall either directly or through her smartphone.

The fundamental research question addressed by this article is how to support the development of flexible interactive applications that span multiple devices and surfaces. From an infrastructure point of view, this requires robust distributed systems that can combine heterogeneous interaction devices, display surfaces and sources of user content; From a visualization point of view, it requires virtual display surfaces that can span multiple physical screens and share content across devices; From an interaction point of view, it requires supporting interaction techniques, such as Rekimoto's pick-and-drop [26], that involve one or more devices. The goal is to create a technical foundation that allows users to *share* both physical and digital resources and interact with them freely.

To address this goal, we introduce *Shared Substance*, a programming framework based on a flexible notion of sharing for developing multi-surface applications. *Shared Substance* enables: sharing of physically connected resources such as mouse pointers and displays [16, 31]; user content, irrespective of how it was created [36]; system elements, such as network connections; and live applications, *i. e.* application state [30] as well as functionality [1].

Shared Substance is based on a novel *data-oriented* programming model where data and functionality are loosely coupled. Our hypothesis is that relaxing this coupling at the programming language level provides the necessary flexibil-

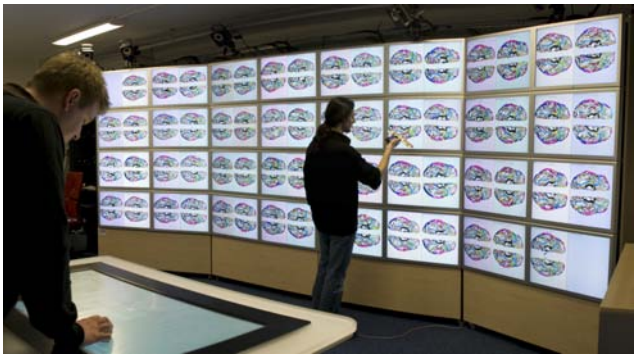

Figure 1. The WILD room: 32-monitor wall display showing 64 brain scans, multi-touch table, motion tracking system.

CHI 2011, May 7–12, 2011, Vancouver, BC, Canada.

ity to support multi-surface interaction. Unlike most previous work that supports sharing either at the architectural level or through specialized components, our sharing mechanisms are close to the programming language so that distribution is as pervasive and transparent as possible for the developer. Our contribution is that data-orientation enables a conceptually simple yet powerful model of sharing that addresses the needs of multi-surface applications.

MOTIVATING SCENARIOS

The main motivation for this work is our exploration of a multi-surface ubiquitous computing environment for scientific discovery called the WILD Room (Fig. 1). The WILD Room consists of a 32-monitor wall display powered by a 16-computer visualization cluster, a multi-touch table and a motion tracking system. It also features wireless devices such as PDAs, smartphones, laptops and tablets that can be added or removed dynamically. Two front-end computers integrate these resources. The ultra-high-resolution wall display has a total resolution of 20480×6400 pixels for a physical size of 5.5 m $\times 1.8$ m. The VICON motion tracking system provides sub-millimeter accuracy and is used to track objects and body gestures in the whole room.

Our users include astrophysicists navigating and annotating very large telescope imagery; biochemists visualizing and interacting with complex molecules; particle physicists examining data from the Large Hadron Collider; and neuroscientists comparing and classifying brain images. They all need to collaborate on and fluidly interact with large, complex data, sometimes using existing applications that were not designed for such an environment. To better understand their needs and explore possible solutions, we have followed a participatory design process. We created a number of concrete interaction scenarios and developed several of them into working prototypes. We present two of these scenarios below, and the corresponding prototypes later in the paper.

Scenario 1: Interacting with heterogeneous data

A group of astrophysicists is studying interstellar phenomena and want to compile a set of high-resolution images, handwritten and typed-in notes, scientific papers, and visual output from their data analysis programs. They bring their laptops and documents to the WILD room and add content to the wall and table, treating them as a large interactive bulletin board: they drag images from their laptops to the wall, ask a distant colleague to send a document to the wall by email, scan their handwritten notes to the table, and display a window from the data analysis tool running on their laptop on the wall. The scientists then cooperatively manipulate, annotate and interact with the content on the wall and table in various ways: by direct touch on the table, using an iPhone or iPad whose position in the room is tracked as both a laser pointer to select content on the display surfaces and as a personal user interface to interact with that content.

Scenario 2: Comparing homogeneous data

A team of neurobiologists study variations in the brain. They want to compare a large number of high resolution 3D brain scans to identify those with a specific pathology. Once the database of brain scans is loaded onto the visualization cluster, they run multiple copies of their analysis software, *Anatomist*[1], side-by-side on the wall to display sixty-four 3D brains. Using a physical model of a brain and a motion-tracked wand, they can control the orientation of all 64 meshes simultaneously. Using the table, they can point at individual brains and rearrange them to, *e. g.*, group similar ones together. For more precise interaction, they can use a laptop and Anatomist's standard interface to remotely interact with some or all of the brains displayed on the wall.

REQUIREMENTS FOR MULTI-SURFACE APPLICATIONS

Our goal is to create a software infrastructure for multi-surfaces applications such as those described above. This infrastructure must fulfill both *application requirements*, which characterize *what* should be possible, and *development requirements*, which address *how* it should be achieved. The requirements below stem from an analysis of scenarios such as those above as well as from our experience with building multi-surface applications from scratch. They are resonant with those identified in, *e. g.*, [1, 10, 14, 19, 30].

Application Requirements

Multi-display strategies: The infrastructure must support arbitrary mappings between physical and logical display surfaces. For instance in Scenario 1 all display surfaces are part of one big, virtual canvas, while in Scenario 2, each screen of the wall holds two display surfaces (one per brain). Mixed configurations are also possible, including several surfaces sharing the same canvas, or some surfaces forming one canvas while others are independent. It should be possible to dynamically reconfigure the role of a surface. In scenario 1 it should be possible to change the role of the table from showing a part of the canvas adjacent to what is shown on the wall, to showing the content of the whole wall, effectively making the table an input device for the wall.

Heterogenous content: The infrastructure must support fetching, displaying and interacting with different types of content. For example, the physicists in Scenario 1 need to juxtapose imagery, scientific data and text documents while in Scenario 2 the neuroscientists manipulate complex 3D scans. Such content comes from different sources, including files, web sites, email attachments and even, as in Scenario 2, from live applications. Content should include not only data, but also the associated methods to display it on various surfaces and interact with it using various devices. Ideally it should be possible to add new types of content at run-time, *e. g.*, when connecting a pen or tablet for handwritten input.

Heterogenous input devices: The infrastructure must support an extensible set of input devices ranging from traditional track pads (multi-touch table, iPads, iPhones) to more exotic devices such as the spatially tracked physical model of a brain in Scenario 2. New input devices can be created by combining existing ones: In Scenario 1, the motion tracking system is combined with an iPhone to create a pointing device with remote control functions available on the touch surface. New input devices and interaction techniques emerge all the time, and it should be possible to easily incorporate them into a multi-surface application.

[1] `http://brainvisa.info`, verified 14 January 2011.

Distributed and parallel interaction: The infrastructure must support interaction techniques adapted to multi-surface applications. In Scenario 1, interaction spans the many surfaces of the canvas, *e. g.*, when dragging an object between physical surfaces. In Scenario 2, a single interaction controls multiple targets simultaneously. In general, multiple users may interact in parallel, possibly with the same data.

Development Requirements

Distributed application model: The infrastructure must support a distributed model for data structures and control flow as well as the allocation and distributed use of physical resources [15, 30, 10]. Since different distribution strategies, such as replication vs. remote invocation, have different performance and robustness trade-offs, developers should have a choice of which strategy to use, including at run-time.

Legacy system agnosticism: The infrastructure should facilitate the integration of legacy systems [6]. The integration may be limited depending on, *e. g.*, the availability of source code or scriptability. In Scenario 1, existing analysis software can be integrated simply at the surface level. In Scenario 2, a deeper integration is needed so that the various replicas of the *Anatomist* application are coordinated.

Low viscosity: The infrastructure should facilitate iteration and experimentation at both design-time and run-time, *i. e.* have low viscosity [23]. At design-time, it is important to be able to test alternative solutions quickly, *e. g.*, for rapid prototyping and participatory design. At run-time, it is important to be able to accommodate unexpected situations so that users are not hindered by the capabilities of the environment. Low viscosity is characterized by flexibility, expressive leverage and expressive match [23].

THE SHARED SUBSTANCE MIDDLEWARE

Shared Substance[2] is a middleware and run-time environment that we have created to support the development of multi-surface applications. It defines a distributed application model based on sharing and provides a set of basic resources for applications, including networking, discovery, file and device access, and integration with GUI toolkits.

Object-oriented approaches to distributed computing, *e. g.*, Distributed Objects [12] and SpeakEasy [21], tightly couple state and behavior, providing a strong conceptual coherence. At the other end of the spectrum, tuple-space based approaches such as EventHeap [15] or One.World [10] decouple state from behavior, resulting in a type of shared memory. Our goal is to provide the best of both worlds: We want to share arbitrary data as shared memory and still be able to associate it with functionality. We believe that the relationship between data and functionality is best addressed at the programming-model level. We therefore take a two-layer approach: The lower layer, *Substance*, is a programming framework based on a *data-oriented* model where state and behavior are loosely coupled; The upper layer, *Shared Substance*, is a middleware that implements the distributed application model and provides the sharing abstractions.

[2] `http://substance-env.sourceforge.net`

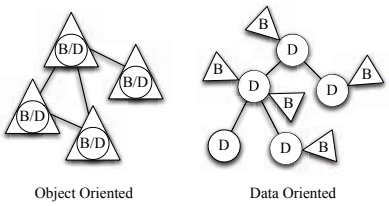

Figure 2. Object-orientation vs. data-orientation. In OO, data and behavior are merged into the Object, which is the primary structuring mechanism of a program. In DO, data and behavior are separated and loosely coupled. Data nodes act as the primary structuring mechanism.

Data-Orientation

Data-Orientation (DO) supports a fundamental separation between data, represented by *Node*s, and functionality, represented by *Facet*s (Fig. 2). Nodes are organized in a tree, where each node is uniquely identifiable by its path . A node may have zero or more facets associated with it. Nodes and facets can be dynamically added and removed at run-time, *i. e.* functionality can be associated to and dissociated from data at run-time. Our motivation for introducing DO is that the hierarchical structure and separation between data and functionality provide a flexible foundation for sharing both application state and behavior in a distributed system.

A DO application consists of a root node with a facet containing the main entry point of the application. For Scenario 1, this facet would create a subtree representing a scene graph consisting of graphical objects. The scene graph would be rendered to the screen by a rendering facet that the main application facet installs on its root. Other facets could be installed on the scene graph itself or on separate subtrees of the application root for, *e. g.*, handling user interaction or adding new content.

Facets resemble aspects in aspect-oriented programming [8]. However, while aspects typically address system-wide concerns such as serialization, facets are local to their node. Facets also resemble dynamic mix-ins that can be added and removed at run-time, but they are more cleanly separated from their data node and from each other than mix-ins.

Substance: A Data-Oriented Framework

Substance is our reference implementation of the data-oriented programming model. Rather than creating a new language from scratch, we used Python, taking advantage of its reflection and meta-programming capabilities. This also gives us access to a rich set of Python libraries.

Substance implements the basic DO concepts, *Nodes* and *Facets*. Nodes contain *values*, the equivalent of OO fields, while facets contain *handlers*, the equivalent of OO methods, *publishers*, to emit events, and *errors*, for exception handling. Nodes and facets are addressed through *paths*, similar to Unix paths or Web URIs. Each node is instantiated with a set of default facets. For example, *CoreStructural* provides functionality for adding and removing children nodes, while *CoreValue* provides local value manipulation.

Programming an application with *Substance* consists in programming its facets. In our reference implementation, facets

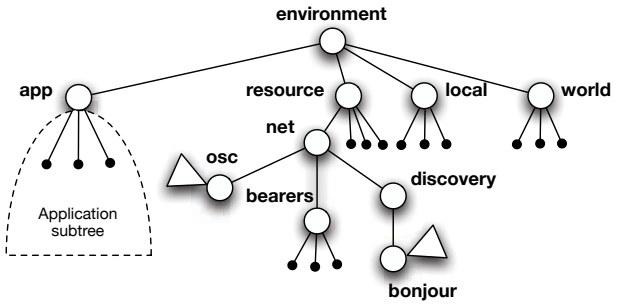

Figure 3. The basic run-time architecture of the *Shared Substance* middleware (circles are nodes and triangles are facets).

are Python objects and can therefore inherit from any Python class. A renderer facet can, *e. g.*, inherit from a Qt Graphics View on Linux or from a Cocoa View on Mac OS X. When a facet is added to a node, *Substance* calls its *instantiate* method and its *destroy* method when it is removed, allowing the developer to respectively set up and clean up values.

Substance supports both reactive (event-driven) and imperative (message-driven) programming: Facets can *listen* to some or all the events *published* by another facet and trigger *handlers*; Facets can also *send* events directly to other facets as messages, using the traditional syntax of a method call. A large part of the implementation of *Substance* uses reactive programming, especially by listening to *CoreValue* and *CoreStructural* events. For example, a rendering facet listens to changes in both the structure and values of a scene graph to update the display.

Shared Substance: A Distributed Application Model

Shared Substance is a distributed application model implemented in *Substance* that introduces the concepts of *environments* and *shares*. *Environments* are uniquely named and remotely discoverable *Shared Substance* processes. A distributed application consists of several environments running on different machines. Each environment can host specific functionality of a distributed application, such as rendering shared data on a particular surface, as well as generic functionality that can be used by multiple distributed applications, such as providing access to the motion tracker in our WILD room. In order for an environment to share data, resources or functionality, it must provide *shares*.

Shares describe subtrees that are publicly available. Shares may be remotely accessed through *mounting*, a strategy similar to remote method invocation (RMI), and *replication*, a strategy similar to shared-state. The distributed application model of *Shared Substance* relies on announcing and accessing shared subtrees and their facets, and replicating and/or mounting them for remote access. Since everything in *Substance* is represented at run-time by trees of nodes and facets, everything (data, resources, functionality, arbitrary segments of an application) may be shared.

Shared Substance defines a standard organization of the nodes of an environment. The root has four subtrees (Fig. 3): *app* and *resource* manage the environment while *local* and *world* manage local configuration and resource sharing.

The *app* subtree is dedicated to the application itself. The *resource* subtree hosts a representation of the locally available physical resources, such as networking and file and device I/O. It is the run-time manifestation of the *Shared Substance* standard library, and may be augmented and configured through a bootstrap mechanism. For example, in order for an environment to support the OSC (Open Sound Control) protocol[3], an OSC subsystem can be added to the bootstrapping sequences and accessed by facets through a path such as *'/resource/net/osc.OSCIO'*.

The *local* subtree holds the local configuration, such as the environment's public name and description, information about the capabilities of the environment, shares and networking resources. Environments and shares are publicly announced for remote dynamic discovery (we currently use Bonjour, but *Shared Substance* is agnostic towards other mechanisms, such as UPnP). The *world* subtree maintains a high-level view of the "world", *i. e.* everything that was discovered by the discovery mechanisms. It provides the abstraction layer for the distributed application model.

A *share* is created by installing a *Sharer* facet on a subtree and providing it with a name and application domain, such as *SceneGraph* and *wild.substancecanvas*. This registers the information about the share in the *local* subtree and announces it on the network. When discovered by other *Shared Substance* applications, it shows up in their *world* subtree.

Accessing a share from a remote environment uses a similar mechanism: Mounting is achieved by installing a *Mounter* facet on a node and requesting to mount the (remote) subtree with the given name and application domain. This creates a proxy node that funnels all events to the remote subtree. Replication is achieved by installing a *Replicator* facet instead. This results in the creation of a local, synchronized replica of the remote subtree. The replica has proxy facets for each facet of the original subtree, which funnel events to the original facets. Replicas can also install new *local facets* locally, *e. g.*, for rendering, which will not be replicated on the original subtree, as well as *shared facets*, which will be available in the remote subtree as well as in any other environment that has mounted or replicated the share.

The rationale for providing two strategies for sharing is based on both performance and conceptual considerations. Mounting is more appropriate when structuring a distributed application in a service-oriented fashion, with a high degree of decoupling between different parts of the application. Since mounting is vulnerable to network latency, *Substance* supports asynchronous calls, with an associated callback method. Mounting is stateless on the sharer side, which means that disappearance of a mounting environment will not affect the sharer. This, however, also means that it is not possible to directly listen for changes in a mounted subtree.

Replication is more appropriate when the goal is to structure the application around shared memory. A replicated subtree is no longer *owned* by the originator, but shared among all replicating environments. Unlike mounting, replication en-

[3]We use OSC extensively in the WILD Room for input devices

courages an event-driven programming style by enabling listening to changes in the replicated subtree. Together with the use of multicast asynchronous communication, this means that replication is less vulnerable to network latency than mounting. Since local copies of the share are stored at each replica, replication is also robust to disappearance of the sharer: Replicas can continue working on their local copy even though it is not synchronized anymore.

Providing two sharing schemes enables developers to more precisely express their intentions with (parts of) an application directly in the application architecture. From a performance point of view, the two strategies should be considered with respect to the access/update frequencies of an application. In general, if both access and updates are rare, the choice of the best strategy is mostly conceptual. If access and/or updates are frequent, our experience shows that replication has better performance, with one exception: If the share holds information that is frequently updated but not frequently accessed, mounting is better than replication.

Instruments: Interaction in Shared Substance

Interaction in *Shared Substance* is based on *instrumental interaction* [2], an interaction model that separates interaction, embodied into *instruments*, from the objects being manipulated. Such separation is particularly suitable to distributed interfaces, as already demonstrated in VIGO [18].

An instrument manages user interaction by mapping input device events to changes in application nodes. Input devices are represented under the *resources* subtree. For example, the VICON motion tracker is represented by a subtree that exposes the most recent position of each tracked object. The instrument typically listens to value updates of the input devices it is interested in and performs changes in application objects that are mounted or replicated. This means that the instrument can run in a different environment than the objects it manipulates. For example, a laser pointer-like instrument can be created by listening to a specific object tracked by the VICON and updating a cursor node in a shared data structure of the application.

IMPLEMENTING THE SCENARIOS

We now illustrate the use of *Shared Substance* with two applications. *SubstanceCanvas* provides a general shared canvas (Fig. 5) for use in Scenario 1. *SubstanceGrise* uses an existing non-distributed application for Scenario 2 (Fig. 6).

SubstanceCanvas

The core of *SubstanceCanvas* is a scene graph shared between a number of environments that provide content, visualize the scene graph and/or support interaction. Each node in the scene graph represents an entity on the canvas, such as an image, a document or a collection thereof, together with information about its position and size.

SubstanceCanvas consists of about 20 individual *Shared Substance* environments (depending on, *e. g.*, how many laptops join the canvas), each with one or more responsibilities: *canvas master*, *renderer*, *content provider*, and *instrument* (Fig. 4). Each environment mounts or replicates the entire scene graph. This ensures that, *e. g.*, moving a picture from

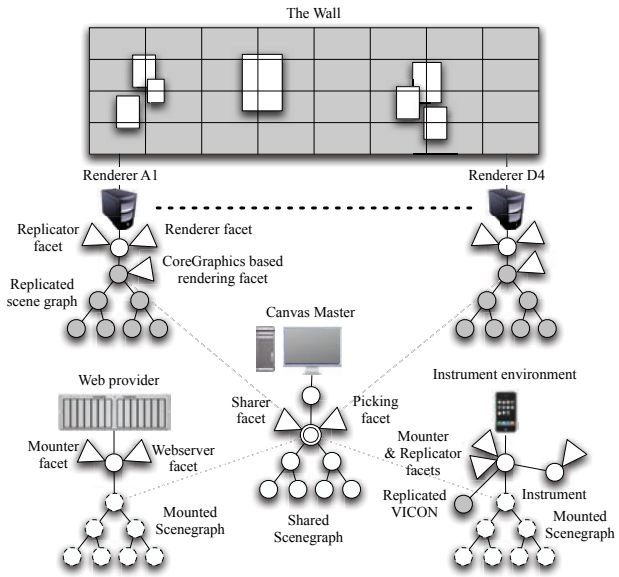

Figure 4. Sharing of the scene graph in *SubstanceCanvas* (circles are nodes and triangles are facets).

one screen to another is just a matter of changing its coordinates on the canvas, and that interaction instruments do not have to worry about physical surfaces.

The scene graph is created and shared by the *canvas master*. The canvas master maps the names of known rendering environments, such as the computers running the wall, to their logical location on the canvas. Multiple environments may render the same portion of the canvas (potentially with different representations), one environment may have several renderers, and any environment may render the entire canvas. When the canvas master discovers a rendering environment in its mapping, it mounts it and asks it to replicate the master scene graph and to render a certain area of the canvas. Since the discovery process is dynamic, rendering environments can come and go, *e. g.*, if the table or a part of the wall is used for something else.

We found that this more centralized approach was more flexible than one where each rendering environment discovers the scene graph, because the latter requires each environment to know its location in the canvas. Our approach makes it easier to change the mapping at run-time. For example, the table initially renders an area logically below the wall, but its rendering facet can be replaced at run-time by one that renders the same area of the canvas as the wall (Fig. 5A). Rendering environments that are not known to the canvas master may still render the scene graph, however they are responsible for knowing which area they render.

The role of a *content provider* is to add new elements to the scene graph. Since it usually does not need to listen to changes in the scene graph, it mounts it rather than replicating it. Different content providers can be distributed among different computers, making it easy to add new content to the canvas. We have created a Web provider to add Web documents to the wall using a simple Web form or bookmarklet,

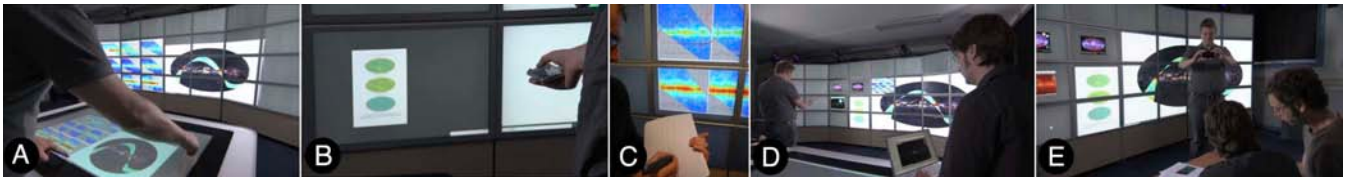

Figure 5. SubstanceCanvas: A) The canvas spans 32 screens and a multi-touch table. B) Interaction on the wall with a motion tracked iPod, C) Live annotation with a digital pen, D) A live window from a laptop is shared with the canvas. E) A photograph is sent to the canvas

an email provider to be used, *e. g.*, with a smartphone, that extracts any attached image and adds it to the canvas, and an application provider that works with any Cocoa-based Mac application to provide live display of its windows.

The *instruments* available in *SubstanceCanvas* support pointing, moving and resizing canvas elements as well as creating annotations. We have created two pointer instruments: one that maps touch positions on a smartphone or tablet to cursor positions in the canvas, and one that combines touch input from the smartphone with positional information from the VICON motion tracker to create a laser pointer-like device. The pointing instrument running on the iPad tablet uses a simplified rendering facet that displays graphical elements as solid shapes, giving a live radar view of the canvas. Each pointing instrument creates its own cursor node in the scene graph and manages visual feedback, *e. g.*, when the cursor hovers over another object. The master canvas provides a facet on the scene graph that instruments can use for picking, *i. e.* translating canvas coordinates into a reference to a graphical element. Finally the annotation instrument adds vector graphics to the scene graph based on input from a live stream of strokes from a digital pen (Fig. 5C).

Both content providers and instruments can run on any device with which the scene graph can be shared, including smartphones, laptops and tablets. Multiple instruments can be active simultaneously: a user in the middle of the room can reposition an image with an iPod-based trackpad while another user resizes another image using a laser pointer, while yet another user pushes a document from the table to the high-resolution wall. In order to create new instruments or content providers, all that is needed is to mount or replicate the shared scene graph, and modify it. This means that new content providers and interaction techniques can be implemented and tested at run-time. In particular, since shared facets can be added at run-time to a replicated graph, new functionality can be added dynamically to the scene graph. For example, a printer environment could install facets on the scene graph that enable a printing instrument running on another machine to print canvas elements.

SubstanceCanvas demonstrates some of the power and flexibility of *Shared Substance*. It uses sharing in a service-oriented fashion, *e. g.*, when discovering and mounting the rendering environments of the scene graph, as well as in a shared memory model, for distributed rendering of the scene graph. It takes advantage of run-time flexibility to support addition and removal of display surfaces, content providers and instruments. Finally it takes advantage of the separation between data and functionality, *e. g.*, when changing the rendering facet for the table at run-time.

SubstanceGrise

SubstanceGrise[4] uses the unmodified *Anatomist* application to render 3D brain scans on the wall. Anatomist can be scripted in Python using a public API and an integrated Python interpreter. We installed *Shared Substance* within Anatomist and created *Shared Substance* environments that can launch, control, and receive events from Anatomist.

Anatomist has two key concepts: *objects* and *windows*. Objects can be 2D bitmaps, 3D meshes, voxel-based volumes, animations, and trees thereof. The user can merge and intersect objects and configure their material properties such as transparency. Windows display 2D and 3D visualizations of the objects they contain along with a 3D cursor used to define cuts of the brain. Each window has a camera defining the distance and angle from which the scan is viewed.

We reflected this data structure in a *SubstanceGrise* node with four subtrees: *objects*, *windows*, *cursors*, and *cameras*. The *objects* subtree holds a URI to load brain data. *Cameras* hold the parameters needed by Anatomist to represent the camera orientation and observer position. *Cursors* hold similar relevant properties for the cursor. *Windows* define the geometry of each window, and contain a reference to a camera, cursor, and their displayed data objects.

The implementation of *SubstanceGrise* consists of 35 environments. A master environment hosts the master application subtree, which it populates with the mapping between brain scans and windows. By sharing this root, it also exports the objects, windows, cursors and cameras.

A separate environment runs for each of the 32 screens and is responsible for displaying a brain scan in each of the two windows it manages. Each environment replicates the master application subtree to access the windows, brain scans, cameras and cursors. Then it installs facets on these objects

[4] *Substance grise* means “grey matter” in French.

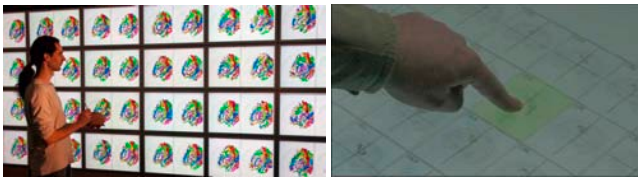

Figure 6. SubstanceGrise: Controlling the camera on 64 different brains (left). Reordering the brains on the multi-touch table (right).

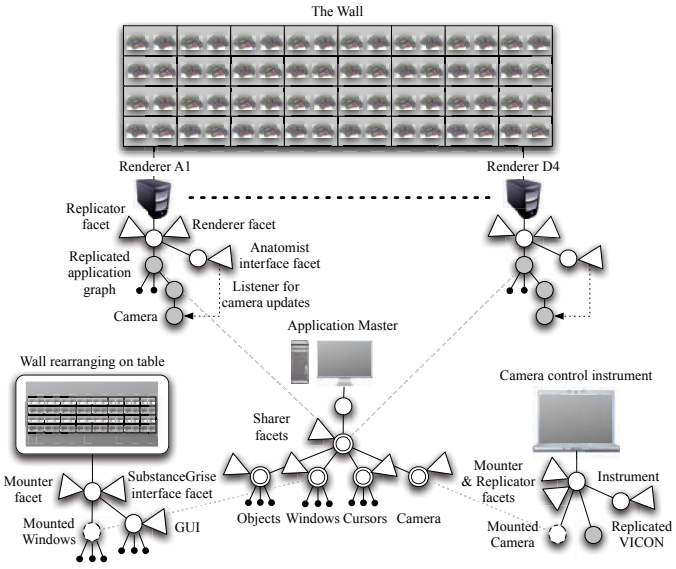

Figure 7. The environments of *SubstanceGrise* (circles are nodes and triangles are facets).

that listen for changes and manipulate their camera or load new objects in their associated Anatomist windows accordingly. These facets are the only part of the *SubstanceGrise* environment that directly interact with Anatomist.

Three instruments control interaction: The tangible brain environment hosts the camera control instrument that lets the neurobiologists change the camera on all 64 brains by pointing on a motion-tracked physical brain model with a motion-tracked pen (much like passive real-world interface props [11]). The brain instrument mounts the camera shared by the master environment and manipulates it according to the notifications coming from the VICON motion tracking system. When the parameters of the camera change, all the rendering environments are notified and propagate the changes to Anatomist. The laptop instrument environment similarly mounts the shared camera and aligns it with the camera of a local Anatomist application running on the laptop.

The instrument running on the table environment uses touch input to control the layout of the brains on the wall. It mounts the windows of the shared graph so that when the user touches two windows, the instrument swaps the corresponding windows in the graph. For feedback, a facet running on each screen of the wall provides the table with snapshots of the graphical state of each screen.

SubstanceGrise demonstrates some of the strengths of *Shared Substance*. First, it shows how to integrate legacy applications. While the embedded Python interpreter of Anatomist made this particularly easy, a similar approach could have been used by wrapping a scriptable application instead. Second, unlike *SubstanceCanvas*, where most environments share the whole scene graph, here specific subgraphs, such as the camera, are shared independently. This makes it easy to create generic instruments that can be used in different applications. For example, controlling the camera through a tangible object only requires a shared camera node. Finally, we took advantage of the data-oriented approach and sharing facilities to smoothly integrate the existing Anatomist application by reflecting it into a set of nodes and facets.

COMPARISON WITH RELATED WORK

Numerous projects have explored interactive applications in multi-surface environments [32]. Distributed rendering tools such as Chromium [13] and Equalizer [7] focus on distributed OpenGL rendering and do not support scene graphs, discovery or sharing. REPO-3D [20], on the other hand, supports these features through COTERIE (discussed below).

Several projects have developed infrastructures for distributed interactive systems. Most of these are event- and data-driven, but these two aspects are typically separated at both the model and implementation levels. A notable, and successful, exception is the Obje middleware for Recombinant Computing [5], which is purely service-oriented, operating on a small set of core functionalities, and relies on services transferring mobile code, including data objects. However, this makes the data-driven part of our data-oriented model, which we use extensively, difficult to implement. Gaia OS [27] (using CORBA) and Sun's Jini [35] (based on Java RMI) face a similar challenge. While Gaia OS provides a file system abstraction (CFS) for sharing resources, as does WebOS [33] (WebFS), this kind of data abstraction does not seem adequate for real-time sharing of, *e. g.*, devices state.

Apart from WebOS, other systems, *e. g.*, CoolTown [17] and XWeb [22], leverage web protocols to provide a data-driven model based on a client-server model. Relying solely on web technology would make *SubstanceCanvas* difficult to implement since the renderers would need to refetch the scene graph representation from the server on each redraw. While XWeb extends this model with an interaction protocol that can be seen as a precursor to SOAP [4] for implementing web services, the problem remains because the server has no way to push changes to the clients once they occur.

The BEACH application model [30] of I-Land [29] relies on shared object spaces for both sharing of and synchronous access to state and functionality. BEACH has a strong and detailed model encompassing most, if not all of the concerns present in distributed interactive environments. However, relying solely on synchronous, direct messaging, would make implementing a shared canvas such as *SubstanceCanvas* cumbersome, as it would require updating or notifying rendering environments individually.

The iRos middleware of iRoom [15] uses Event Heap [15], an event-driven framework that uses a shared tuple-space for events, allowing services to subscribe to events on the heap in a distributed blackboard fashion. iRos provides a Data Heap abstraction for sharing data, supporting a more data-driven mode of operation, while iCrafter [25] is a service-sharing framework implemented on top of Event Heap. Similarly, One.World [10] uses a service-oriented model where services directly exchange asynchronous events and provides a separate shared tuple-space [9] for sharing state and data. These approaches provide many of the same services as *Substance* and *Shared Substance*, but we believe that the

close connection between data and events in *Substance* results in a conceptually simpler and more coherent approach. For instance, iRos requires design-time decisions about accessing an element through events, data, or both. With our approach, this decision can be postponed until run-time.

Finally, sharing at the language level, such as Java's Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) [34] simplifies the distributed application programming model by mimicking local constructs such as method calls. The problem however is that local and distributed applications are inherently different [28], so completely hiding the distribution aspects is not a good solution in general: distribution should be explicit, and the developer should be able to choose between data-driven and message/event-driven control flow. COTERIE [19] follows this model, with a language-level approach similar to ours. It supports both fully distributed *Shared Objects* and remotely accessed *Network Objects*, similar to our replication and mounting strategies. However, each component must be programmed as one or the other, limiting the flexibility at run-time. Also, COTERIE relies on centralized repositories for discovery, which can be a bottleneck.

DISCUSSION

Shared Substance is a middleware based on a data-oriented programming model that provides a sharing abstraction to develop multi-surface applications. While it is not as complete as, *e. g.*, existing GUI toolkits, the scenarios used in this article have illustrated its power and flexibility. *Substance* and *Shared Substance* were developed through a series of participatory design and development workshops, where we assessed whether developers used to object-oriented programming would be able to use the concepts effectively. In the rest of this section we discuss the lessons learned from these workshops and from the development and use of *Substance* and *Shared Substance*, in particular with respect to the requirements identified at the beginning of the paper.

Separating Data from Functionality

The main thrust of our approach is the fundamental separation between data and functionality. How easy is it for developers to adapt to this new programming model? The workshops showed that at first, developers used data-orientation in an object-oriented way, attaching one facet to each node. But once given examples, they were quickly able to use other patterns such as multiple facets per node or a facet controlling a whole subtree, and take advantage of this new flexibility. The workshops also helped us identify counter-intuitive or heavy syntax, *e. g.*, explicit declaration of path objects or explicit unpacking of events, which led to a much better integration with the host language (Python).

Enforcing a data-oriented approach also required a scheme to integrate existing libraries and applications. We used an ad hoc strategy where each relevant library was wrapped by a set of *Substance* subtrees and facets that integrate smoothly with the environment, in particular through sharing. The granularity of the integration is left to the developer. For example, our integration of Apple Core Graphics is done through a single facet, while that of the Anatomist application uses a subtree to represent the state of the application and facets to interface with its functions. We found that using subtrees to represent state that can be shared and listening to changes in the values and/or structure was particularly powerful, for example to integrate input devices such as the VICON tracker or communication protocols such as OSC or HTTP. Overall, this approach supports the *legacy system agnosticism* and *heterogenous input devices* requirements.

The requirement for *heterogenous data content* is supported by the open data model and by the fact that any *Substance* node can be extended with new values and children at run-time. For example, in *SubstanceCanvas*, content providers can add arbitrary data to the scene graph, even though a specific renderer for this data is not present. As long as they include meta-data such as position, shape and dimension, our generic renderers can display the shape of the objects.

When replicating a subgraph, an environment can install local facets directly on the data, *e. g.*, for rendering, and these facets can be mixed with facets that are proxies to remote environments. This way, functionality can be kept local to the data it is logically connected to, even in the presence of replication, while still maintaining the loose coupling. For example, in *SubstanceCanvas*, the renderer facet on the iPad tablet only displays the object shapes, while a different facet renders the content with full detail on the wall. This loose coupling also makes it possible to add and change functionality at run-time. For example, in *SubstanceCanvas*, the rendering of the scene graph on the table can be changed on the fly by replacing the rendering facet. This approach supports the requirement for *different multi-surface strategies*.

Sharing, Replication and Mounting

The hierarchical structure of *Substance* encourages the developer to organize an application in logically consistent subtrees so that specific parts of the application can be shared independently. *Shared Substance* directly supports the requirement for a *distributed application model* through the concept of replicating or mounting a shared subtree.

How does this approach address the trade-off inherent to distributed systems between transparency and explicit control of the distribution? While previous approaches, *e. g.*, Event Heap [15] and COTERIE [19], tend to require explicit control, *Shared Substance* provides a good level of transparency. Once the sharing facets (sharer, mounter, replicator) have been added to a subtree, there is almost no difference between a local and shared subtree. The example applications show the value of a unified mechanism for sharing functionality, data and physical resources. For example, input from the VICON motion tracking system is stored in a subtree that is replicated by instrument environments. This supports the requirement for *distributed and parallel interaction*.

The choice between mounting and replication is, however, important. Initially we assumed that each application would use one or the other, but our experience showed that both were useful in a single application. Mounting is primarily used to provide a service-oriented approach, *e. g.*, mounting a renderer and telling it to replicate a given scene-graph, whereas replication is mainly used for shared memory, *e. g.*, to share a scene-graph among renderers. The workshops

showed that while developers quickly understood and took advantage of sharing, they needed more time and experience to assess the best trade-off given the limitations of our implementation. One main drawback is that mounting currently does not support listening to value changes, which can be useful in a service-oriented approach.

Viscosity

Low viscosity is characterized by flexibility (the ability to make and test changes rapidly), expressive leverage (achieving more with less), and expressive match (conceptual distance between problem and solution) [23]. *Shared Substance* provides a high level of flexibility because of the ability to use shared data as the glue between the various environments making up an application. For example, in *SubstanceGrise*, individual components such as cameras and windows are shared separately, making it easy to experiment with different ways of rearranging windows on the wall. We started with a simple environment with a command-line interface to rearrange the brains on the wall, and iteratively refined it into the interactive table solution.

Flexibility also stems from the data-oriented model of *Substance*, and in particular the ability to easily replace facets, even at run-time. For example, we used *SubstanceCanvas* to create an application displaying content as an array. To manipulate the array, we reused the table interface from *SubstanceGrise* and simply replaced the facet that interfaced with the Anatomist window by one that interfaced with the *SubstanceCanvas* scene graph.

Supporting sharing as a fundamental characteristic of *Shared Substance* provides expressive leverage to the developer, since little effort is needed to create distributed applications. For example, once the data structure of the Anatomist application is mirrored in the *SubstanceGrise* application graph, it can be shared without knowing exactly how it will be used. *Substance* also provides expressive leverage through the separation of data and functionality: The relationships between a concept, *i. e.*, data, and its uses, *i. e.*, functionality, are more fluid than in object-oriented programming. For example, new facets can be added to existing data, even at run-time, to support new functionality.

A potential weakness of our approach is the lack of a detailed architectural model. Unlike, *e. g.*, BEACH [30], we provide a very simple distributed application model but rely on the developer to create the right architecture. This could negatively affect expressive match since the lack of guidance could result in poor choices and overly complex and/or rigid solutions. Our workshops have shown that developers understand the concepts and, with some practice, use them properly. While this still needs to be validated on a larger scale, it is also likely that architectural patterns will emerge as we gain experience with developing more applications.

Other Engineering Concerns

Distributed applications typically need to address security, performance, scalability and concurrency issues. We have not addressed security in *Substance* because our applications did not call for it. A potential approach is to to implement security policies within the core facets, *e. g.* by requiring certain credentials to modify the structure or access the nodes, or authenticating the sender of an event based on its path.

Performance and scalability have not been the main focus of our work. Our current implementation works well for several dozen environments extensively sharing subtrees on a local network. For example, displaying the canvas across the entire wall is smooth and responsive without explicit synchronization. Performance is likely to degrade with hundreds or thousands of environments, however this is not the type of applications we are targeting. Also, there is ample room for improving the implementation, *e. g.*, by using C instead of Python in critical components.

We have used an optimistic approach to concurrency, where the last update always wins. This has worked well except in a few specific cases. An area for future work is to implement one of the well-known concurrency control algorithms to ensure that updates to the tree are consistent across replicas.

Other Application Areas

The range of applications targeted by *Shared Substance* is much wider than the examples that illustrate this article. Given our current experience, we are confident that *Shared Substance* is well adapted to small and medium-size distributed interactive applications such as multiple users interacting with a shared, public display [24] or mobile collaborative games [3]. In both examples, the content of the display or the state of the game can be replicated and modified by the various users while the run-time flexibility supports the addition and removal of users as well as features.

CONCLUSION

We have presented an infrastructure facilitating the development of multi-surface applications. We have shown that the combination of a novel programming framework called data-orientation and a middleware supporting powerful sharing abstractions provides a solid foundation for multi-surface applications. We have described *Substance* and *Shared Substance*, our reference implementation. We have illustrated its use through two real-world examples and have shown how it meets the key requirements for multi-surface interaction.

While this work was motivated by the need to develop applications for the WILD Room, we are confident that *Shared Substance* can be used as a general purpose ubiquitous computing framework. For example, a sensor network could use dynamic discovery and subscriptions to access sensor data through polling (using mounting) or notification (using replication). The discovery mechanisms and dynamic capabilities also make it well-suited for on-the-fly collaboration, *e. g.*, to share content in small meetings.

Future work includes using *Shared Substance* to further explore interaction in distributed multi-surface environments. At the technical level, we are building on *SubstanceCanvas* to create a GUI toolkit supporting multi-surface rendering and interaction instruments. At the conceptual level we are exploring forms of sharing that can scale to very large applications. Finally, we are continuing our workshops with developers and software designers to assess how *Shared Substance* is adopted and should be further developed.

ACKNOWLEDGEMENTS

This work was partially supported by the French ANR grant "iStar" (#2007-TLOG-009-03) and by the Digiteo / Région Île-de-France grant "WILD" (#2008-25D). We thank the workshop participants for their time and suggestions.

REFERENCES

1. G. Banavar, J. Beck, E. Gluzberg, and J. Munson. Challenges: an application model for pervasive computing. In *Proc. ACM Mobile Computing and Networking*, MobiCom '00, 266–274, Jan 2000.
2. M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. ACM Human Factors in Computing Systems*, CHI '00, 446–453, 2000.
3. M. Bell, M. Chalmers, L. Barkhuus, M. Hall, S. Sherwood, P. Tennent, B. Brown, D. Rowland, S. Benford, M. Capra, and A. Hampshire. Interweaving mobile games with everyday life. In *Proc. ACM Human Factors in Computing Systems*, CHI '06, 417–426, 2006.
4. F. Curbera, M. Duftler, R. Khalaf, and W. Nagy. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, Jan 2002.
5. K. Edwards, M. Newman, J. Sedivy, and T. Smith. Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Trans. Computer-Human Interaction (ToCHI)*, 16(1):1–44, 2009.
6. K. Edwards, M. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: recombinant computing and the speakeasy approach. In *Proc. ACM Mobile Computing and Networking*, MobiCom '02, 279–286, 2002.
7. S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Trans. Visualization and Computer Graphics*, 15:436–452, 2009.
8. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.
9. D. Gelernter. Generative communication in linda. *ACM Trans. Prog. Lang. and Syst. (TOPLAS)*, 7(1):80–112, Jan 1985.
10. R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Trans. Computer Systems (TOCS)*, 22(4):421–486, 2004.
11. K. Hinckley, R. Pausch, J. C. Goble, and N. F. Kassell. Passive real-world interface props for neurosurgical visualization. In *Proc. ACM Human Factors in Computing Systems*, CHI'94, 452–458, 1994.
12. P. Homburg, L. V. Doorn, M. V. Steen, A. Tanenbaum, and W. D. Jonge. An object model for flexible distributed systems. *Vrije Universiteit*, 1995.
13. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proc. ACM Computer Graphics and Interactive Techniques*, SIGGRAPH '02, 693–702, 2002.
14. B. Johanson and A. Fox. The Event Heap: a coordination infrastructure for interactive workspaces. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, HotMobile '02, 83– 93, 2002.
15. B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–74, Jan 2002.
16. B. Johanson, G. Hutchins, and T. Winograd. PointRight: experience with flexible input redirection in interactive workspaces. In *Proc. ACM User Interface Software and Technology*, UIST '02, 227–234, Jan 2002.
17. T. Kindberg and J. Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443–456, Jan 2001.
18. C. Klokmose and M. Beaudouin-Lafon. VIGO: instrumental interaction in multi-surface environments. In *Proc. ACM Human Factors in Computing Systems*, CHI '09, 869–878, 2009.
19. B. Macintyre and S. Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proc. ACM User Interface Software and Technology*, UIST '96, 83–94, Jan 1996.
20. B. Macintyre and S. Feiner. A distributed 3d graphics library. In *Proc. ACM Computer Graphics and Interactive Techniques*, SIGGRAPH '98, 361–370, Jan 1998.
21. M. Newman, S. Izadi, W. Edwards, J. Sedivy, and T. Smith. User interfaces when and where they are needed: an infrastructure for recombinant computing. In *Proc. ACM User Interface Software and Technology*, UIST '02, 171–180, 2002.
22. D. Olsen, Jr, S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal interaction using XWeb. In *Proc. ACM User Interface Software and Technology*, UIST '00, 191–200, 2000.
23. D. R. Olsen, Jr. Evaluating user interface systems research. In *Proc. ACM User Interface Software and Technology*, UIST '07, 251–258, 2007.
24. T. Paek, M. Agrawala, S. Basu, S. Drucker, T. Kristjansson, R. Logan, K. Toyama, and A. Wilson. Toward universal mobile interaction for shared displays. In *Proc. ACM Computer Supported Cooperative Work*, CSCW '04, 266–269, 2004.
25. S. Ponnekanti, B. Lee, A. Fox, and P. Hanrahan. Icrafter: A service framework for ubiquitous computing environments. *Proc. ACM Ubiquitous Computing (UbiComp'01)*, 56–75, Jan 2001.
26. J. Rekimoto. Pick-and-drop: a direct manipulation technique for multiple computer environments. In *Proc. ACM User Interface Software and Technology*, UIST '97, 31–39, Jan 1997.
27. M. Roman and R. Campbell. Gaia: Enabling active spaces. *Proc. ACM SIGOPS European Workshop (EW 9)*, 229–234, Jan 2000.
28. A. Rotem-Gal-Oz. Fallacies of distributed computing explained. 2010.
29. N. Streitz, J. Geißler, T. Holmer, and S. Konomi. i-LAND: an interactive landscape for creativity and innovation. In *Proc. ACM Human Factors in Computing Systems*, CHI '99, 120–127, Jan 1999.
30. P. Tandler. The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments. *Jal of Systems and Software*, 69(3):267–296, 2004.
31. P. Tandler, T. Prante, C. Müller-Tomfelde, N. Streitz, and R. Seinmetz. Connectables: Dynamic coupling of displays for the flexible creation of shared workspaces. In *Proc. ACM User Interface Software and Technology*, UIST '01, 11–20, 2001.
32. L. Terrenghi, A. Quigley, and A. Dix. A taxonomy for and analysis of multi-person-display ecosystems. *Personal and Ubiquitous Computing*, 13(8), Jan 2009.
33. A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proc. ACM High Performance Distributed Computing*, HPDC '98, 1998.
34. S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, Jan 1997.
35. J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 76–82, Jan 1999.
36. D. Wigdor, H. Jiang, C. Forlines, M. Borkin, and C. Shen. WeSpace: the design development and deployment of a walk-up and share multi-surface visual collaboration system. In *Proc. ACM Human Factors in Computing Systems*, CHI '09, 1237–1246, Apr 2009.