In: Proc. of Ubiquitous Computing 2003 (Ubicomp'03). Seattle, Washington.

An Architecture and Framework for Steerable Interface Systems

Anthony Levas, Claudio Pinhanez, Gopal Pingali, Rick Kjeldsen, Mark Podlaseck, Noi Sukaviriya

IBM T.J. Watson Research Center
Hawthorne, New York, USA
http://www.research.ibm.com/ed/

Abstract. Steerable Interfaces are emerging as a new paradigm used in realizing the vision of embodied interaction in ubiquitous computing environments. Such interfaces steer relevant input and output capabilities around space, to serve the user when and where they are needed. We present an architecture and a programming framework that enable the development of Steerable Interface applications. The distributed multi-layer architecture provides applications with abstractions to services of several novel components – for instance, steerable projection, steerable visual interaction detection, and geometric reasoning. The programming framework facilitates integration of the various services while hiding the complexity of sequencing and synchronizing the underlying components.

1 Introduction

Weiser [1] characterizes "good" technology as being invisible and highlights the trend in ubiquitous computing towards "invisible" computers. The quote, "Invisible technology stays out of the way of the task - like a good pencil stays out of the way of writing", stresses the importance of technology allowing the user to focus on the task at hand, and allowing them to take advantage of tacit and contextual knowledge, while unencumbered by the technology itself. As a principle for building invisible interfaces, he remarked, "the unit of design should be social people in their environment, plus your device". As the focus of interaction moves away from the desktop focusing on the individual, his relationship to the physical world and the objects that inhabit it, new technologies will be needed to afford effective interaction.

Dourish [2] has elaborated the notion of *embodiment*[1], and argues that Tangible [3] and Social Computing [2] are aspects of the same perspective he calls Embodied Interaction, which we believe forms a compelling basis for studying and building interactions in ubiquitous environments. In his view "Embodied interaction provides conceptual tools for understanding how the interface might move into the background

[1] "embodiment …denotes a form of participative status, …and is about the fact that things are embedded in the world and the ways in which their reality depends on being embedded."

ceptual tools for understanding how the interface might move into the background without disappearing altogether." [2]. Rehman et al. [4] argue that the trend towards invisible computers have left the user without meaningful cognitive models of system function and behavior. They propose a solution can be found in the use of the "Many, many [input/output] displays" Mark Weiser [1] predicted would be available in ubiquitous environments of the future. Rehman addressed this problem using head mounted displays and cameras to present information to users in a contextually appropriate manner. We share the common view expressed by Dourish and Rehman that it is important to afford natural interaction to the user when and where they need it in the environment without disappearing altogether. However, we feel that wiring the environment with many displays is difficult and that wearing head mounted gear or carrying around devices is encumbering. Our work focuses on a new paradigm we are developing called Steerable Interfaces (SI) [5]. The main principle here is to use devices that can steer relevant input and output capabilities to the user in the environment, when and where they are needed. This obviates the need to carry or wear devices or to outfit the environment with many displays or other devices.

Steerable Interface technology enables the computer interface to dynamically reach into the environment and interact with the user based on where he is and what he needs. For example, an interactive display may be projected onto a nearby table that allows the user to manipulate a projected control (e.g. slider) to change some feature of the environment. Physical objects may be dynamically annotated with projected information regarding their use and endowed with behavior through steerable input provided by visual gesture recognition devices. This would allow the user to manipulate them in a tangible manner to achieve a desired goal. Instead of having many interactive displays of varying sizes, projected displays may be resized to suit the needs of the interaction, perhaps small for private notification or large for interacting with colleagues. A projection could also concurrently simulate many interactive displays on a given surface by partitioning its overall space into smaller displays that have the appearance of being separate.

The computer no longer needs to be thought of as a device that is viewed through a fixed screen found in a static location, but as a device that can "reach" into our "real" world to interact with us. One may view this as embodying the computer such that it has a dynamic presence in our world and can interact with us in a goal directed manner. Nothing, however, prevents us from creating "bad" interfaces using this technology other than grounding our work in principles developed through the study of Embodied Interaction.

Pinhanez [6] describes the Everywhere Displays (ED) Projector, a projector with an associated rotating mirror that enables images to be projected onto virtually any surface while correcting for oblique projection distortion. Kjeldsen et al. [7,11] discuss adding Steerable Interaction capabilities to the ED, through a steerable vision system that can recognize hand gestures. Pinhanez et al. [8] and Sukaviriya et al. [9,10] describe several applications that highlight the use and advantages of SI's. Pingali et al.[5] have defined the concept of SI and described its characteristics and articulated the enabling technologies. This extends the use to the acoustic domain through the use of steerable microphone arrays [12] for input and steerable audio output [13]. Lai et al., [14] have developed a prototype of an adaptive, personalizable workspace, called BlueSpace, using this technology. This workspace application has received significant publicity in the media to date.

This paper focuses on an architecture and programming framework that supports the development of Steerable Interfaces. This paper is organized as follows. Section 2 describes the requirements that drove the architectural design as well as the resulting architecture. Section 3 discusses the requirements and design of the software development framework. Section 4 illustrates the use of the Java Application Development Environment and highlights productivity gains. Section 5 demonstrates system use through an example application. Section 6 concludes with a discussion on what we have learned and directions for future work.

2 System Architecture

Our primary goal here is the development of an architecture that facilitates the development of effective tools and techniques for building and delivering Steerable Interfaces, as well as the exploration of new modes of interaction for ubiquitous computing environments. This ranges from the design and development of specific technologies, for example in projection and visual interaction, as well as the combination of technologies to form new interaction capabilities.

Analysis of our primary goal resulted in the following set of requirements for an architecture that enables Steerable Interfaces:

- Support communication among the multiple hardware and software components used in Steerable Interface systems.
- Insure that components support the definition of an interaction that is independent of the location where the interaction is delivered. Allow the same interaction to be delivered to any available location in the environment.
- Provide a clear abstraction between function and implementation of components to facilitate experimenting with new ways to deliver underlying function without affecting applications that use the component.
- Insure components operate independently as well as work together.
- Facilitate the easy addition of new components to extend functionality.
- Afford both asynchronous and synchronous operation of components from layers above.
- Enable development using different languages and development approaches.
- Support a distributed architecture.
- Insure adequate performance.

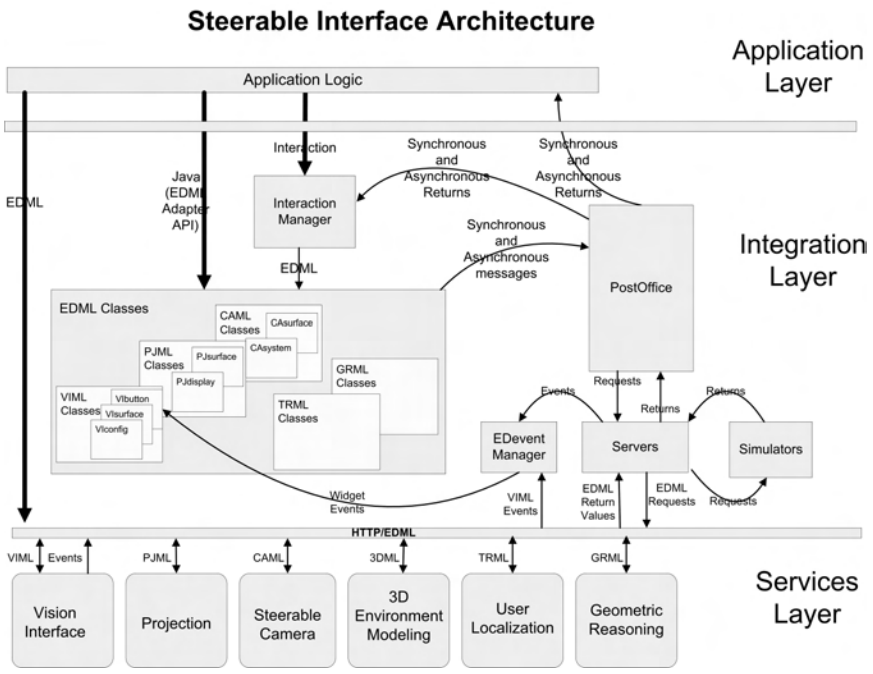


Figure 1. An architecture for steerable interfaces.

Figure 1 shows our architecture for Steerable Interfaces. It is a three-tier architecture composed of a Services Layer, an Integration Layer and an Application Logic Layer, as explained below.

2.1 The Services Layer

The Services layer consists of a set of modules that represent independent components of a Steerable Interface system. Each of these modules exposes a set of core capabilities through an HTTP/XML Application Programming Interface (API). Modules in the Services Layer have no "direct" knowledge or dependence on other modules in that layer. The modules share a common XML language called the Everywhere Displays Markup Language (EDML) along with a specialized dialect for communication with each module in this layer. Applications written in any language can directly communicate with each module in this layer using EDML, as indicated by the first arrow on the left in figure 1, labeled EDML. EDML allows us to explore developing applications using a variety of different approaches by providing language independence. Multiple modules of each type can be instantiated and operated at the same time.

The core EDML definition includes commands for establishing communication with a module, starting and stopping their respective services, and commanding or querying modules. Basic EDML action commands fall into 3 logical pairs: Use/Release are used for definition/allocation and de-allocation of objects (e.g. Buttons, Images, etc.). Set/Get are used for setting or retrieving values of objects and Activate/Deactivate for activation and deactivation of “Used” (allocated) objects.

All modules in the services layer respond to XML commands asynchronously, i.e., once an XML command is received through an HTTP “POST” message, the HTTP communication is first completed before the request is processed. The sender tags each message and upon completion of the command the component returns an XML message along with the identifying tag, indicating either that it has successfully completed or that an error was encountered. We chose this approach as opposed to using a Remote Procedure Call (RPC) approach such as RMI, CORBA or SOAP to facilitate asynchronous communication where the caller does not wait for the completion of the command. Brumitt et al. [15] and Arnstein et al. [16] have elaborated on problems related to RPC style communication in ubiquitous computing environments and have chosen a similar asynchronous approach. Also, using the HTTP protocol allows us to more easily implement a distributed system and configure the machines on which a service runs.

The Services Layer has six types of components that have been implemented to provide an initial basis for applications that use Steerable Interfaces. The Vision, Projection and Steerable Camera interfaces afford steerable visual input and output, while the 3D Environment Modeling, User Localization and Geometric Reasoning components enable tracking of individuals in the space and reasoning about individuals and the spatial relationships they have with interaction surfaces in the environment. These last three components have been included because they are especially useful in achieving adaptation of the interface to user and environmental context. Brumitt et al. [17] have articulated a common view regarding the importance of geometry and geometric reasoning in their EasyLiving [18] project. In our system, the tracking, modeling, and reasoning components provide critical contextual information that allows us to direct the interactive display onto the appropriate surfaces that are most effective for the interaction required. For instance, occlusion, which is often a problem in steerable interfaces, can be detected by geometrically reasoning about the position of the user and the projected surface. If occlusion is determined to occur, the projected interaction can be redirected and the content adapted. We will show how this is particularly useful in the example in Section 6.

The **Projection** component EDML API, called PJML, provides the declarative representation of the extents of display images along with their spatial arrangement in a known frame of reference. The basic PJML action commands enable the instantiation, activation and deactivation of this declarative information. The projection surfaces and associated real world parameters are known only to this component and are also referred to by the application symbolically by name. This leaves the responsibility of mapping the declaratively defined visual widgets to relevant surfaces, as well as the intrinsic implementation, completely to this component. Two basic widgets have currently been defined for this component. An *Image* is basically a bitmap that can be projected onto a surface. A *Stream* is a rectangular region defining a portion of the display buffer on the machine that is running the Projection component. This region can be dynamic visual imagery that is then mapped onto the surface designated via PJML.

The **Steerable Camera** component EDML API, called CAML exposes an interface that allows the application to steer the camera to a named location and adjust all the relevant parameters, such as, zoom, focus and gain to access video data. As in the case of the other modules this is a general-purpose component that can be used by applications to capture and use video data as is required.

The **Vision Interface** component EDML API, called VIML, provides two very important functions. First, it enables the declarative description of the widgets of a specific visual interface interaction along with their spatial arrangement in a known frame of reference. The basic VIML action commands enable the instantiation, activation and deactivation of this declarative description. The interaction surfaces and associated real world parameters are known only to this component and are referred to by the application symbolically, by name. This leaves the responsibility of mapping the declarative interface description to relevant surfaces completely to this component. Second, it provides a mechanism for returning events to applications that relate to the users manipulation of a widget - such as triggering a button press. All the necessary usage and environmental details are hidden from the application developer, who is concerned only with the arrival of an event associated with a specific widget. The component implementation is free to change allowing us to explore alternative approaches for recognizing gestures.

We have currently defined and implemented three vision interface widgets, a *Button*, a *Slider*, and a *Track Area*. Buttons are simple visual objects that are triggered when "touched". Sliders allow a user to adjust a continuous single dimension parameter by moving their finger along a designated rectangular region. Track Areas allow a user to adjust a continuous two dimension parameter (X,Y) by moving a finger within a rectangular region. The Projector component can endow widgets with a visual appearance by projecting imagery that coincides with the position of the widget in the camera's field of view, but this is not a necessity. Widgets can be tangible objects, such as real Buttons, or Sliders, or even pictures of widgets. This component's responsibility is to provide the basic functions for recognizing gestures in a ubiquitous computing environment.

Figure 2 graphically illustrates the mapping process that occurs when an application combines the function of the Projection component, the Vision Interface component and the Steerable Camera component to create an *interactive display* in the real world. The Definition process highlights that a specific interaction has a single declarative display and interaction area. During the Mapping phase the Projection and Steerable Camera components steer their associated devices to the surfaces named in the defini-

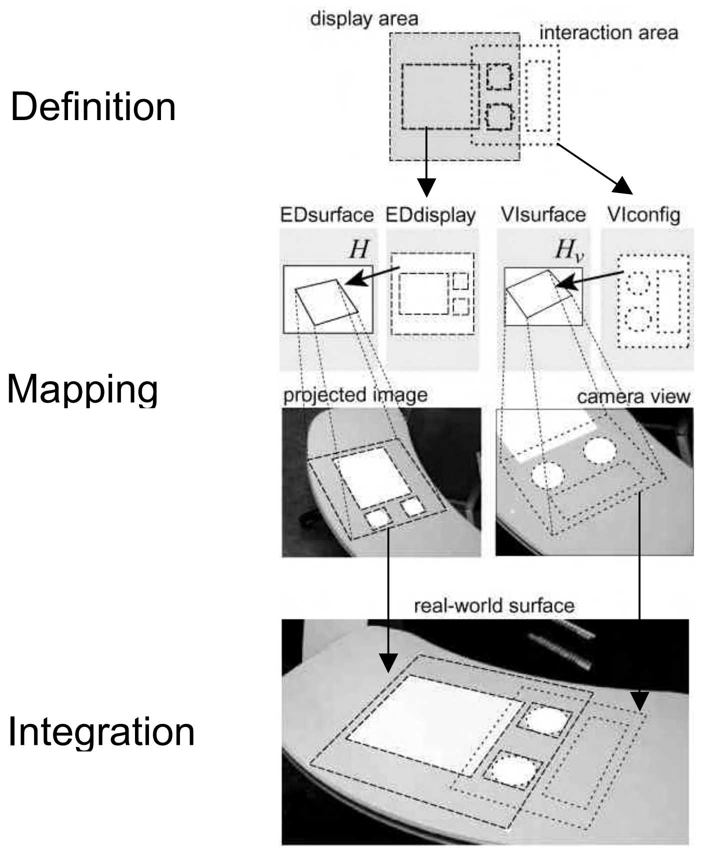


Figure 2. The process of mapping the declarative display and interaction area definitions onto real world surfaces.

tion. *H* and *Hv* are the homogeneous transformations that define the mapping of the display and interaction areas to projection and interaction surfaces respectively. These transformations for defining surfaces are derived from a calibration that is first performed on the respective host machines upon which the Vision Interface, the Steerable Camera, and Projector components are running.

The **3D Environment Modeling** component is a modeling toolkit for creating and manipulating an environment model to support steerable interfaces. It is in many ways a simplified version of standard 3D modeling software. It supports basic geometric objects built out of planar surfaces and cubes and allows importing more complex models. However, the toolkit provides additional objects such as projectors and projection display surfaces and annotation capabilities that are specifically required for steerable interfaces. Almost every object in the model can be annotated. This makes it possible to attach semantics to objects such as optical properties of a surface and its preferred usage. The toolkit stores the model in XML format, with objects as tags and annotations as attributes. This format allows the model to be easily defined and manipulated by applications in the architecture. The modeling toolkit is also designed to be accessible to the Geometric Reasoning component described below.

The **User Localization** component keeps track of the 3D position of a user's head in relation to the environment model. In our current implementation, we use multiple cameras with overlapping views to triangulate and estimate this position. Our system tracks user position to an accuracy within a few inches and operates at video frame rate. This allows us to reason quite effectively about the appropriate surface for displaying the interface and about user occlusion of the display, as discussed above. The EDML API for the User Localization component is called TRML, and provides a basic structure for queries about the user's presence, position, and orientation in the environment.

The purpose of the **Geometric Reasoning** component is to facilitate intelligent selection of the appropriate display and interaction surfaces based on criteria such as proximity to the user and non-occlusion of the surface by the user or by other objects. Applications or other modules can query the geometric reasoning engine through the EDML API, called GRML. The API currently supports two types of queries. The first type of query is a property look-up action. The geometric reasoning engine returns all the properties of a specified display surface given the location of the user in the environment. The user location is typically obtained by the application from the User Localization component described above. Other possibilities include simulated user positions for planning and testing purposes. In the second type of query, the reasoning engine receives a user position and a set of criteria, specified as desired ranges of display surface properties, and returns all surfaces that satisfy those criteria. Properties of a display surface include:

1. Physical size of the display surface.
2. Absolute orientation angle between the surface normal and a horizontal plane.
3. Distance between the center of the user's head and the center of a surface.
4. Position of the user relative to the display surface, defined as the two angles to the user's head in a local spherical coordinate system attached to the display surface. This indicates if the user is to the left or to the right of a surface.
5. Position relative to the user, defined as the two angles to the display surface in a local spherical coordinate system attached to the user's head.
6. Occlusion percentage, defined as the percentage of the total area of the surface that is occluded with respect to a specified projector position and orientation.
7. An occlusion mask, which is a bitmap that indicates the parts of a display surface occluded by other objects in the model or by the user.

2.2 The Integration Layer

The Services Layer described above has been designed to meet some of the important requirements listed at the beginning of section 2. Each component in the Services layer has a very specialized function in a particular aspect of a ubiquitous computing environment. Through EDML, the layer provides one level of abstraction between function and implementation of components to facilitate replacing the underlying components that realize a function without affecting applications that use the component.

However, *coordination* of these components to deliver new functionality is beyond the scope of the Services Layer. Hence, we defined an Integration Layer that has a built-in set of classes to facilitate the integration of the underlying components in the Services Layer. We decided that JAVA would be a powerful programming language to use in defining this set of classes for developing ubiquitous applications. Our goals here were to provide a set of basic classes that could be used by an Application Framework to provide a set of concepts that would facilitate development. The requirements we derived for this layer are:

- Provide a JAVA based API for all EDML objects and associated commands
- Facilitate communication with the services layer by providing a set of specialized servers to handle message input and output as well as events returned
- Handle event propagation back to the appropriate widgets
- Provide simulation capability for each service, enabling development without the need to be connected to the precious resources of the space
- Handle synchronous and asynchronous message passing and callbacks as well as coordination or synchronization of sets of asynchronous calls

Figure 1 illustrates the classes that were built to satisfy these requirements. The **Servers** block consists of a set of classes for generating servers (as well as specific server instances) that handle serialization of messages sent to each component, insuring that the order of arrival is the same as the order of transmission. This is quite important in cases where one command to a component must precede another. In addition, these servers also handle messages returned from components signaling success or failure in execution of the requests and events returned from the vision interface for propagation to the application.

The **PostOffice** class allows the JAVA based EDML commands to either run synchronously, that is, wait for the return result of the requested command, or asynchronously, with the ability to get a callback upon completion of the command. The PostOffice uses a simple mechanism, similar to the concept of "Return Receipt" common in real post offices. The idea is that each request is tagged with a unique ID. If a synchronous call is requested, the calling thread issues a *wait*, and proceeds only after the post office *notifies* it of the receipt of a "return receipt" that has the unique ID. Similarly, asynchronous calls would just proceed, but would have a callback method called upon receipt of the "return receipt" that had the appropriate unique ID. Note that messages could share a common "return receipt" ID, in which case a thread could wait for the completion of a set of asynchronous commands, a very useful and frequently used feature.

The **EDML Classes** allowed a JAVA program to easily construct all the objects defined by EDML and to issue synchronous or asynchronous calls to the components in the Service Layer. Visual interaction widgets are subclasses of JAVA swing classes. For example, buttons are a subclass of *JButton*, so they would inherit all their capabilities, again a very useful feature, which we will discuss in the next section. Similarly, display widgets are inherited from relevant swing classes. For example, an Image is a subclass of *Jlabel.* To invoke the behavior of a Button the Event Manager simply needs to call the *doClick* method of the Button. The advantage of this approach is that the same functionality can be triggered through several means.

For example, consider a media player application that can be moved to different surfaces in space, allowing the user to play, stop, rewind etc., by touching projected buttons. Our approach allows this media player application to be controlled either through gestures recognized through the camera based Vision Interface, or simply through an alternate on-screen GUI.

To facilitate development, we have also incorporated Simulators to the integration layer framework. **Simulators** allow application development to proceed without the need to actually be in the physical space. Each component has a corresponding simulator class that could be used to simulate receipt of messages and returns that would normally be generated by the actual component. Also, an event simulator is available for transmitting simulated events, which is again, very useful in developing and debugging an application. Several levels of debug traces can be switched on and off, providing a complete trace of the system.

The second arrow from the left, labeled JAVA (EDML Adapter API) in Figure 1 indicates the path that a JAVA application could take by calling the set of JAVA EDML classes. Taking this direct path to the EDML classes, however, can be tedious and requires coordination of numerous calls and a significant amount of code to create simple applications. Note that an application that generates an interactive display would have to individually coordinate all of the components in the layer below. In particular, rapid prototyping can be hard to achieve using the EDML JAVA classes directly. To address this problem, we have developed a framework to simplify the development of Steerable Interface applications, as we will shortly discuss in Section 3.

2.3 The Application Logic Layer

The Integration Layer facilitates communication with the Services layer and coordination of multiple services, as described above. The final layer in our architecture is an Application Logic Layer where an actual application is instantiated, resulting in the instantiation of the corresponding classes in the Integration Layer. The framework for developing such an application is discussed below. This framework was used in a JAVA Application Development Environment that enables graphical composition of the GUI to significantly simplify the whole process of development.

3 Steerable Interface Application Programming Framework

We developed a programming framework consisting of a set of classes that conceptually and practically simplify the task of application development. The current framework consists of over 130 classes including classes in the Integration layer. However, the framework enables applications to be written using as few as 4 or 5 classes. We now outline the major classes that simplify application development.

The primary class for creating an interactive display is called **Interaction**. It contains all the necessary information required by the Vision Interface, the Projector and the Steerable Camera to create a declarative definition of display widgets (e.g. images and streams) as well as vision interface widgets (e.g. buttons, and sliders) on a given surface. In addition, it contains the procedural information defining the behavior triggered by user interaction. It contains at least 20 instances of EDML objects that specify attributes such as the projection and vision surfaces, the container object that hold the display and interaction widgets, along with the widgets themselves, and many other parameters. It basically encapsulates everything needed for one projected interactive dialog. It would take many JAVA calls to instantiate an Interaction, and requires detailed knowledge of layout to provide coordinates for these objects, and to fill out the many parameters. This process is akin to building a traditional GUI, only more complex due to the coordination of a greater number of components. Fortunately, Application Development Environments (ADE) have solved this problem before, through the use of graphical GUI construction tools. Using the Composition Editor (a JAVA bean editor) in our ADE, an Interaction can be completely defined graphically. Stubs for actions to perform are also generated in the process and need to be fleshed out to define the behavior. An application developer would use this tool to generate a number of interactions for a particular application. We will describe the development process in Section 4.

The main driver for an application is the **Application** class. The set of Interactions used by the application are added to an Application instance. The Application instance contains the logic of an application and can maintain the relevant application state required in making any decisions. An Application can define a new **SequenceManager** that is responsible for sequencing the order that Interactions are presented or instantiate one from an existing set of available managers. A simple SequenceManager may call a method in the Application or the current interaction to compute the next Interaction to instantiate.

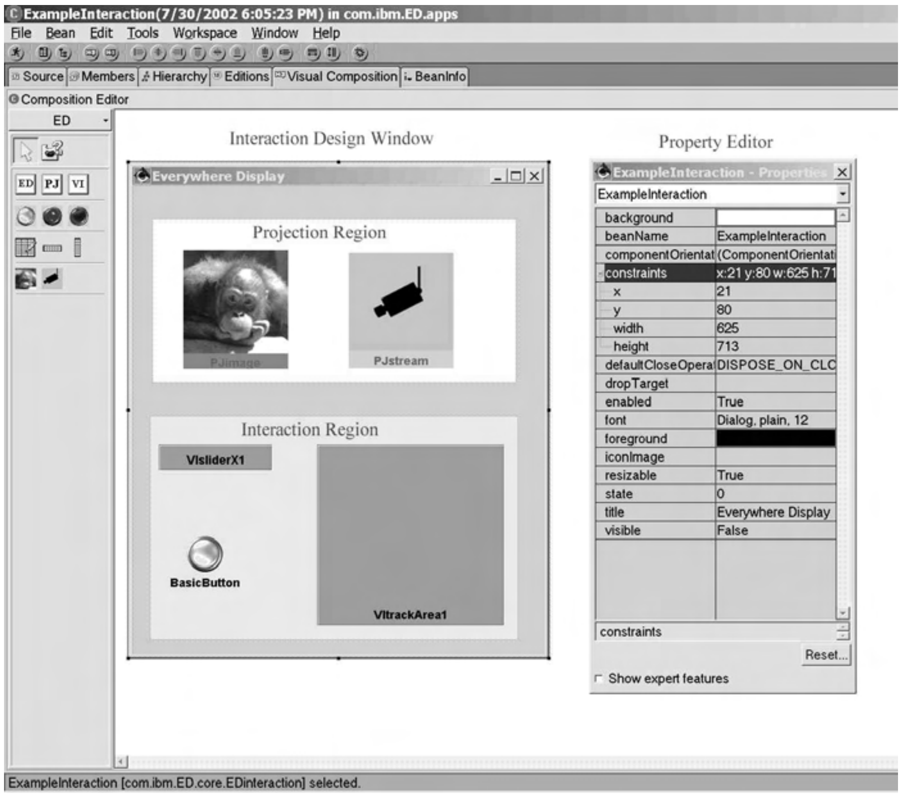
Once the Application is ready to instantiate an Interaction, it automatically passes it to the **InteractionManager,** which is responsible for orchestrating all the EDML calls that are necessary for bringing the interaction to life. Figure 1 shows this path that completes a description of the three ways that we have to generate Steerable Interfaces. Using this approach requires knowledge of only a few Framework classes and simplifies the task of application development, especially when using the Composition Editor described in Section 4.

A few other classes are useful for tracking a user in the environment and triggering behavior based on position. A **Tracker** class is provided that monitors the position of the user in the environment at a specified frequency. It uses the Observer/Observed pattern to publish location information to all subscribers. A **Zone** class defines a region in space and is an Observer of the Tracker class. It has several methods that fire when a user enters a Zone or leaves a Zone. This class currently has 2 subclasses that implement **Circular** Zones and **Rectangular** Zones. Zones can be added to Applications to automatically trigger behavior appropriate to a location in space.

Finally, a class called **TrackerGUI** allows simulation of the movement of a user in the environment through a mouse-based GUI and a simple graphical representation of the physical space. In this case, the **Tracker** simply gets its tracking information from the TrackerGUI as opposed to the User Localization component. We stress the importance of simulation in frameworks for ubiquitous computing environments. Not only does it allow development to proceed without requiring the resources of the space, but it also helps in debugging the overall complex solution and track down bugs in the underlying components. Much of the Integration Layer and application Development Framework was developed before the Services Layer components were completed and debugged. Also, debugging and integration of the whole architecture was significantly simplified. In Section 5, we will describe how all the classes described above were used to implement an interactive advertising display one might encounter in a retail store.

4 Java Application Development Environment (ED-JADE)

Based on the framework described in Section 3, we developed a Java Application Development Environment that enables graphical composition of Steerable Interface applications. Figure 3 shows the Composition Editor, available in our ADE, that we used to create Interactions. The left-hand column of the GUI contains JAVA Swing classes we have sub-classed. The center region is the design window. The dialog on the design window is a visual representation of the interaction we are building. An application developer would populate this dialog with Images and Streams that would be available in the dialog by simply clicking on the appropriate widget and dragging it onto the dialog. Similarly, the developer would select interaction widgets from the palette and place them in the appropriate location. A property editor is available for editing all the relevant fields in the interaction. The property editor, for instance, could be used to define the image that is projected for the Button or the Track Area. In addition, a default display surface could be specified indicating where this dialog should


ExampleInteraction(7/30/2002 6:05:23 PM) in com.ibm.ED.apps
File Bean Edit Tools Workspace Window Help
Source Members Hierarchy Editions Visual Composition BeanInfo
Composition Editor
ED
ED PJ VI
Interaction Design Window
Property Editor
Everywhere Display
Projection Region
PJimage
PJstream
Interaction Region
VIsliderX1
BasicButton
VItrackArea1
ExampleInteraction - Properties
ExampleInteraction
constraints
Reset...
Show expert features

background	
beanName	ExampleInteraction
componentOrientat	(ComponentOrientati
constraints	x:21 y:80 w:625 h:71
x	21
y	80
width	625
height	713
defaultCloseOperat	DISPOSE_ON_CLC
dropTarget	
enabled	True
font	Dialog, plain, 12
foreground	⬛
iconImage	
resizable	True
state	0
title	Everywhere Display
visible	False

ExampleInteraction [com.ibm.ED.core.EDinteraction] selected.

Figure 3. The Composition Editor in our JAVA Application Development Environment.

appear in the environment. Using this approach a developer can quickly define the widgets and their spatial relationships without resorting to code.

5 An Example Application

Figure 4 illustrates one example application that was developed using the framework described in the previous sections. This application is a very simple demonstration of the use of a steerable interface in a department store of the future. This example interface combines steerable projection, gesture recognition, user tracking, geometric reasoning, and environment modeling capabilities described in Section 2. In the top left picture an advertisement is presented to the general public as they pass by and tries to lure them toward the display by indicating the word “GIFT” in a large font. As a shopper approaches, the application detects the presence of the shopper and adds some smaller text which is now viewable from this new position saying “A GIFT of beauty” just for you (Fig. 4 – top left). As the shopper approaches even closer a new

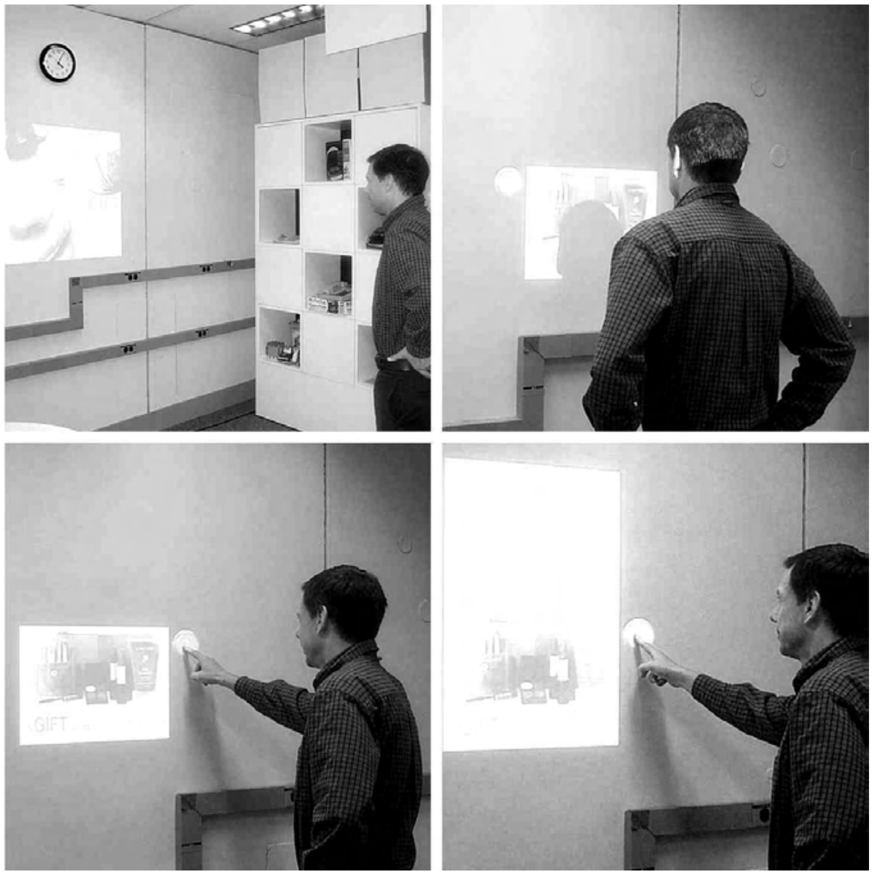


Figure 4. An interactive display demonstration for a product in a retail store.

image is presented showing the potential free gifts available, along with a button that can be pressed to obtain a detailed description of the gifts possible (fig. 4 – top right). However, in this situation the shopper is in a position that occludes the display. The application reasons about the occlusion and moves the display to a new unoccluded surface near the customer and adapts the interface by positioning the button to a location that is easily accessible (Fig. 4 – bottom left). The shopper, being interested in seeing more details, presses the button and is presented with an additional description of what they can get for free (Fig 4 – bottom right).

The whole application above was developed with less than 200 lines of code using the application framework and the ADE described in the previous section. In this case, since the interaction could be handled within a single surface (about a 4 foot square projection area) only one Interaction needed to be defined. To implement this application, all we had to do was to instantiate an Application class object and define and add an Interaction containing three rectangular zones. A simple observer class was written and added to the Tracker. Its function was specifically to detect occlusion based on customer position and adapt the image appropriately.

To deploy the application in a space, the Vision Interface, Projection and Steerable Camera components were calibrated to one single surface of our laboratory environment using the calibration GUI tools of the respective components. The position of the surface was also entered in the database of the 3D Environment Modeling component. The application runs continuously, once started, luring those that pass by in our lab to interact with it.

6 Discussion and Conclusions

We have presented an architecture and a programming framework for Steerable Interfaces and illustrated their utility through an example application. We believe that our architecture successfully meets the requirements of a Steerable Interface system architecture as outlined in Section 2. The XML approach to the Services Layer results in a distributed architecture that allows the various components to exist on different machines. This approach also facilitates replacement of individual components and exploration of alternate means of achieving the same functionality. The Integration Layer enables coordination, sequencing, and synchronization of the modules provided by the Services Layer, and provides means of simulating these Services. Finally, the Application Programming Framework allows easy development of applications while hiding the complexity of the Services and Integration layers.

Several developers in our group and in other groups have used this architecture and programming framework in different environments. The architecture has enabled eloquent demonstrations of the new styles of interaction offered by the Steerable Interfaces paradigm. However, we recognize that this is still a first attempt at an architecture to support this new paradigm. Below, we summarize some of our observations based on the experience gained thus far in using it.

We have found that the design of the Services Layer and associated communication approach described here has indeed enabled the exploration of several alternative approaches to application development. For example, some of our colleagues have used JavaScript in conjunction with Microsoft's XML/HTTP ActiveX objects to communicate with this layer. Others have used MacroMedia's Flash to link to this layer, while a group at a collaborating university has used JAVA to directly generate XML. Some members of our group rely mostly on developing applications using the JAVA EDML API, while others use the application development Framework extensively.

The extensive use of simulation has been very important in allowing development of the components to proceed in parallel as well as being able to develop applications with out requiring a lot of time in the actual physical space or use of associated resources like the projector and camera. It has been our experience that application development for ubiquitous computing environments in general will have to rely heavily on such simulations.

We realize that adaptability is a critical concern for Steerable Interface systems. The ability to effectively use contextual information to determine the most appropriate surface for interaction as well as adapting the content based on this context is crucial. The support for geometric modeling and reasoning in our architecture has allowed us to develop several applications that exhibit such adaptability. In particular, we are able to address one of the most important issues in Steerable Interfaces, namely occlusion.

We have worked with other colleagues to develop a prototype workspace environment called BlueSpace[14] using the framework described in this paper. This environment has been demonstrated extensively at several research centers around the world. Recently, this system was demonstrated extensively for nine days at CeBIT 2003 in Hanover, Germany and proved to be quite robust. The Steerable Interface technology was very well received by the business community, the media and by every day people that attended – a positive indication of the potential of this paradigm.

In future, we plan to continue to evolve this architecture to support the growing needs of Steerable Interface applications. At the Services Layer, we plan to incorporate the features of the latest projection devices, as this is an area that is advancing quickly. Adding steerable audio input and output functionality for voice and sound are another area we plan to explore. This will focus our attention on issues related to multimodality in our architecture. At the Integration Layer, we also plan to extend and test the architecture to address the coordination of numerous steerable input and output units over large spaces. Support for remote monitoring, control, and diagnosis are other challenging architectural issues. Other issues of interest go beyond the architecture to the underlying science and technologies for Steerable Interfaces. Simplifying or minimizing the need for calibration is a very important area to ensure that these kinds of applications abound. Understanding human perception of surfaces and objects that are being projected upon will allow the selection and adaptation of surfaces based on their characteristics as well as the projection characteristics. Finally, looking at novel gestural interaction paradigms that are better matched with ubiquitous computing environments is a topic we will pursue.

References

1. Weiser, M.: Bulding Invisible Interface. Keynote: *User Interface, Systems, and Technologies (UIST)* conference, November, 1994, slides: http://www.ubiq.com/hypertext/weiser/UbiHome.html
2. Dourish, P. Where the Action Is. MIT Press, Cambridge Massachusetts (2001)
3. Ishii, H., Ulmer, B.: Tangible Bits:Towards Seemless Interfaces between People, Bits and Atoms. In: Proc. of CHI'97. Atlanta, Georgia. 1997. pp. 234-241

4. Rehman, K., Stajano, F., Coulouris, G.: The Interfacing with Invisible Computers. NordiCHI, October 19-23, 2002, Arhus, Denmark.
5. Pingali, G., Pinhanez, C., Levas, A., Kjeldsen, R., Podlaseck, M., Chen, H., Sukaviriya, N.: Steerable Interfaces for Pervasive Computing Spaces. PerCom 2003, Ft. Worth, Texas
6. Pinhanez, C., The Everywhere Displays Projector: A Device to Create Ubiquitous Graphical Images. Proc. Of Ubicomp 2001, Atlanta, Georgia, September, 2001.
7. Kjeldsen, R., Pinhanez, C., Pingali, G., Hartman, J., Levas, A., Podlaseck, M., Interacting with Steerable Projected Displays. International Conference on Automatic Face & Gesture Recognition, May, 2002, Washington, D.C.
8. Pinhanez, C., Kjeldsen, R., Levas, A., Pingali, G., Podlaseck, M., Sukaviriya, N.: Using Computer Vision in Steerable Applications. Submitted to ICCV 03
9. Sukaviriya, N., Podlaseck, M., Kjeldsen, R., Levas, A., Pingali, G, Pinhanez, C.: Embedding Interactions in a retail Store Environment: The Design and Lessons Learned. To appear in INTERACT03 Ninth IFIP TC13 International Conference on Human-Computer Interaction, September 2003
10. Sukaviriya, N., Podlaseck, M., Kjeldsen, R., Levas, A., Pingali, G, Pinhanez, C.: Augmenting a Retail Environment Using Steerable Interactive Displays. In Proceedings of CHI 2003 Conference on Human Factors in Computing Systems, April 2003.
11. Kjeldsen, R., Levas, A., Pinhanez, C.: Dynamically Reconfigurable Vision-Based User Interfaces. 3rd Intl. Conf. on Computer Vision Systems, April 2003, Graz, Austria.
12. Silverman, H., Patterson, W., Flanagan, J. and Rabinkin, D. "A Digital Signal Processing System for Source Location and Sound Capture by Large Microphone Arrays", In Proceedingsof International Conference on Acoustics Speech and Signal Processing ICASSP 97, Munich, Germany, April 1997.
13. Pompei. F. J.: The Audio Spotlight. Journal of the Audio Engineering Society, Vol. 47, pp. 726-731, October 1999.
14. Lai, J., Levas, A., Chou, P., Pinhanez, C., Viveros, M.: BlueSpace: Personalizing Workspace Through Awareness and Adaptability. In International Journal of Human-Computer Studies, vol. 57(5): 415-428, 2002.
15. Brumitt, B., Meyers, B., Krumm, J., Kern, A., Shafer, S.: EasyLiving: Technologies for Intelligent Environments, In HUC, pages12-29, 2000.
16. Arnstein, L. et al., Systems Support for Ubiquitous Computing: A case Study of Two Implementations of Labscape. Proceedings of the First International Conference on Pervasive Computing, 2002, Springer-Verlag, Germany
17. Brumitt, B., Krumm, J., Meyers, B., Shafer, S.:Ubiquitous Computing & the Role of Geometry. In IEEE Personal Communications, August 2000.
18. Shafer et al. The Easy Living Intelligent Environment System. In Proc. of CHI Workshop on Research Directions in Situated Computing, April 2000.