

Dynamically Reconfigurable Vision-Based User Interfaces

Rick Kjeldsen, Anthony Levas, Claudio Pinhanez

IBM T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598
fcmk, levas, pinhanez@us.ibm.com

Abstract. A significant problem with vision-based user interfaces is that they are typically developed and tuned for one specific configuration—one set of interactions at one location in the world and in image space. This paper describes methods and architecture for a vision system that supports dynamic reconfiguration of interfaces, changing the form and location of the interaction on the fly. We accomplish this by decoupling the functional definition of the interface from the specification of its location in the physical environment and in the camera image. Applications create a user interface by requesting a configuration of predefined widgets. The vision system assembles a tree of image processing components to fulfill the request, using, if necessary, shared computational resources. This interface can be moved to any planar surface in the camera's field of view. We illustrate the power of such a reconfigurable vision-based interaction system in the context of a prototype application involving projected interactive displays.

1 Introduction

Vision-based user interfaces (VB-UI) are an emerging area of user interface technology where a user's intentional gestures are detected via camera, interpreted and used to control an application. Although the recognition of human gesture and action has been the topic of many workshops and conferences [1-3], and the focus of much of our previous work [4, 5], the problem of design and implementation of these applications as well as the integration of computer vision has received, comparatively, less attention. Most real-life vision interface systems incorporate the vision system as a module that is hard-coded to operate under a fixed set of circumstances. In this paper we describe a system where the application sends the vision system a description of the user interface as a *configuration* of widgets (describing *What* the interface is). Based on this, the vision system assembles a set of image processing components that implement the interface, sharing computational resources when possible. To change the interaction, a new interface description can be sent to the system at any time.

The architecture also provides for the deployment of an interface onto different real-world planar surfaces. The parameters of the surfaces where the interface can be realized are defined and stored independently of any particular interface. These include the size, location and perspective distortion within the image and

characteristics of the physical environment around that surface, such as the user's likely position while interacting with it. *When* the application requests a interface be activated on a particular surface (that is, *Where* the interaction should happen in the environment) the system propagates the surface parameters down the assembly of image processing components that implements that interface.

By explicitly decoupling the information describing the characteristics of *Where* an interface happen in an environment, i.e., surface-specific information, we facilitate (1) the porting an application to a new environment where the interaction surfaces are different; (2) the use of one surface for multiple applications; and (3) the use of the same interface on multiple surfaces.

These issues are very important in our current work that investigates steerable, projected interactive user interfaces, as described later in this paper (see also [6]). However, the framework presented in this paper should be seen as a way that vision-based applications can easily adapt to different environments. Moreover, the proposed vision-system architecture is very appropriate for the increasingly common situations where the interface surface is not static (as, for instance in the cardboard interface described in [7]), when a pan/tilt camera is used to make an interface follow the user (as in [8]), or when the camera is attached to the user as in applications involving augmented reality or wearable computers(see [9]).

The main contribution of this paper is the system architecture for the support of these *dynamically reconfigurable vision-based user interfaces*, both from the application point of view and in the inner workings of the vision system.

2 Basic Elements of Dynamically Reconfigurable VB-UIs

We start the discussion of our framework by describing three primitive concepts: configurations, widgets, and surfaces.

2.1 Configurations and Widgets

In our framework, a vision-based user interface is composed of a set of individual interaction dialogs referred to as *configurations*. Each configuration is a collection of interactive *widgets*, in a structure similar to how a traditional window-based application is defined as a set of dialog windows, each containing elements such as scroll bars, buttons and menus. In the case of a VB-UI, each widget provides an elemental user interaction, such as detecting a touch or tracking a fingertip. Widgets generate events back to the controlling application where they are mapped to control actions such as triggering an event or establishing the value of a parameter. Some of our earlier work describes the individual widget types we use and how they are implemented [10, 11]. Here we will focus on how they are dynamically combined to create a user interface.

In addition to defining the widgets, a configuration specifies a *boundary* area that defines the configuration coordinate system. The boundary is used during the process of mapping a configuration onto a particular surface, as described later.

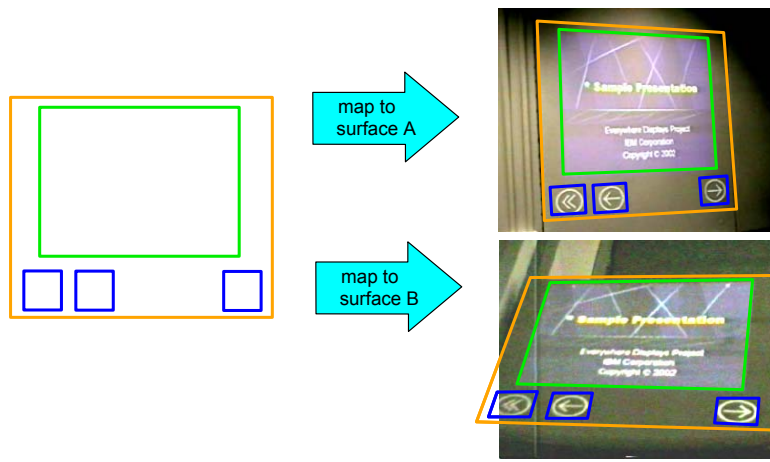


Fig. 1. Mapping a configuration onto two different surfaces.

2.2 Surfaces

An application needs to be able to define the spatial layout of widgets with respect to each other and the world, as that is relevant to the user experience, but should not be concerned with details of the recognition process, such as exactly where these widgets lie within the video image. To provide this abstraction we use the concept of named interaction *surfaces*. A surface is essentially the camera's view of a plane in 3D space.

When a configuration is defined, its widgets are laid out using the coordinate system defined by the boundary area. A configuration is mapped to a surface by warping that coordinate system into the image with a perspective transformation (homography). When the configuration is activated, the region of the image corresponding to each widget is identified and examined for the appropriate activity, which in turn will trigger events to be returned to the application. Figure 1 shows a configuration with three buttons (blue squares) and a tracking area (green rectangle) being mapped onto different surfaces. The process of determining the homography and establishing other local surface parameters is described in the next section.

3 Architecture of a Dynamically Reconfigurable Vision System

In order to efficiently support dynamic reconfiguration of vision-based interfaces, a flexible internal architecture is required in the vision system. In addition, the vision system must support operations that are not visible to the application, such as calibration, testing, and tuning. This section will describe this internal architecture.

In our system, each widget is represented internally as a tree of *components*. Each component performs one step in the widget's operation. For example the component tree of a "touch button" widget is circled in figure 2. There are components for

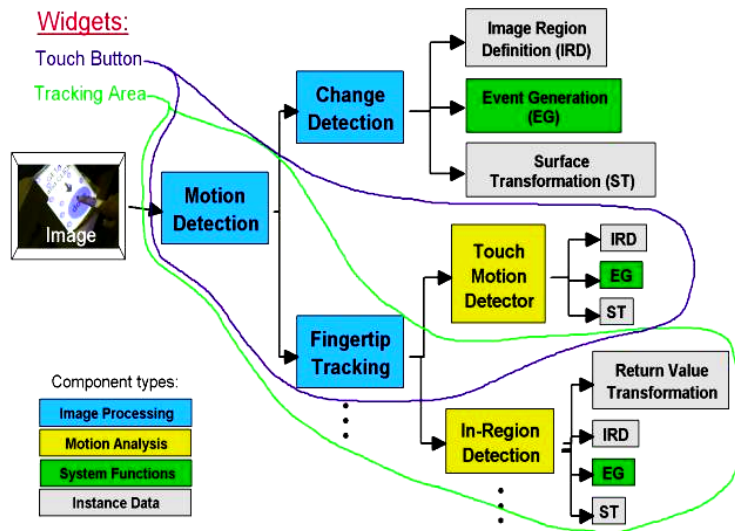


Fig. 2. Tree of components for two widgets.

finding the moving pixels in an image (Motion Detection), finding and tracking fingertips in the motion data (Fingertip Tracking), looking for touch-like motions in the fingertip paths (Touch Motion Detection), generating the touch event for the application (Event Generation), storing the region of application space where this widget resides (Image Region Definition), and managing the transformation between application space and the image (Surface Transformation).

Information is passed into the trunk of this tree and propagates from parent to child. During image processing, images are passed in. In this example, the Motion Detection component takes in a raw camera image and generates a motion mask image for its child components. Fingertip Tracking takes the motion mask and generates a path for the best fingertip hypothesis. Touch Motion Detection examines the fingertip path for a motion resembling a touch inside the image region of this button. When it detects such a motion, it triggers the Event Generation component. A similar structure is used by the “tracking area” widget, also circled in figure 2. Because of the structured communication between components, they can easily be reused and rearranged to create new widget types with different behavior.

3.1 Shared Components

When an application activates a configuration of widgets, the vision system adds the components of each widget to the existing tree of active components. If high-level components are common between multiple widgets, they may either be shared or duplicated. For example, if there are multiple Touch Button components, they can share the motion Detection and Fingertip Tracking components, or each may have its own copy. The advantage of shared components is that expensive processing steps

need not be repeated for each widget. Unfortunately this can sometimes lead to undesirable interactions between widgets, so the application has the option of specifying that these components be shared or not as needed.

A good example of the trade-offs of shared components is when using touch-sensitive buttons. If there are multiple buttons active at one time, these buttons generally share the Motion Detection component. When the Fingertip Tracking component is shared, however, the behavior of the widgets can change. Recall that the Fingertip Tracker component tracks fingertip hypotheses within a region of the image. If this component is shared by more than one button, these widgets will both use the same fingertip hypothesis, meaning that only one of them can generate an event at a time. This may be desirable in some circumstances, say when implementing a grid of buttons, such as telephone keypad. In other circumstances, however, the application may not want activity in one button to prevent operation of another, so the widgets should each have their own fingertip tracker.

3.2 Communication and Control

When components are combined in a tree, widgets lose their individuality. However, it is still necessary to have a mechanism able to send information to and from widgets both individually and in groups (e.g. all widgets in a configuration). Information is propagated down the tree by Posting typed data to the root nodes. Data is retrieved from components in the tree by Querying for some data type. Both Post and Query use a fully qualified *address* including Configuration Name and Widget Name, either of which can be “all”. As Post and Query data structures flow through a component, the address and data type of the structure are examined to determine if it should be handled or ignored by that component.

For example, during operation, image data addressed to all widgets is posted to the root components of the tree. As the data flows from parent to child, some components, such as the Motion Detector, may choose to modify the image before they post it to their children. Others, like the Fingertip Tracker, may create a new type of data (in this case a fingertip path) and post that to their children instead.

3.3 Surface Calibration

Applications identify surfaces by name, but each surface must be calibrated to determine where it lies in the video image. A surface is calibrated by identifying the points in the image that correspond to the corners of a configuration’s boundary. These points can be located either manually or automatically, and then are saved with the surface. When a configuration is mapped to a surface, the point pairs for each boundary corner are posted to the component tree. Each widget has a Surface Transformation (ST) component that computes a homography from the four point-pairs, and then uses it to convert the widget’s configuration coordinates into image coordinates. The other components of the widget query the ST component to determine what image region to examine.

3.4 Vision System Parameters

In order to get the best performance from the vision system, a number of internal parameters can be adjusted. We keep these parameters hidden from the application so that the application need not be concerned with the specifics of visual recognition, and so the internal implementation of the widgets can change without requiring changes to the application.

The system maintains a local record of all configurations and surfaces that have been defined, and parameters are maintained independently for each one. The developer of an application can manually adjust (and test) the parameters from the vision system local GUI. When a configuration is mapped to a surface and activated, the parameters of both the configuration and the surface are retrieved.

Configurations maintain parameters for each widget component, allowing control of image-processing aspects such as sensitivity to motion. This allows the application designer to adjust each configuration for different recognition characteristics. For example, one configuration may need a higher recognition rate at the expense of a higher false positive rate, while in another a high false positive rate may not be acceptable. Surfaces maintain parameters about the physical environment, such as where a user is typically located with respect to the surface during an interaction, which can be used by the widgets during processing

4 An XML API for a Dynamically Reconfigurable VB-UI System

To create a VB-UI an application must define the What, When and Where of each interaction. Defining What and When is similar to developing standard non VB-UI applications. One or more configurations must be defined, specifying the spatial layout of the widgets in each. The sequence of configurations (and of the non-UI aspects of the application) must be defined as a function of the events returned from the widgets and the application state. Unique to VB-UI interactions, the Where of each interaction must also be defined, meaning on which surface a configuration is to be displayed.

To give the application the needed control we have defined an API based on a dialect of XML we call VIML (Vision Interface Markup Language). VIML defines a set of visual interface objects and methods. Three basic objects are: *VIsurface* for defining attributes of a surface; *VIconfiguration* for defining widgets, their spatial relationships and elaborating their behavior; and *VIevent* for communicating events, such as a button press back to the application. In this paper we are concerned only with three methods for VIconfigurations and VIsurfaces: “Set”, used for setting values of objects; and “Activate/Deactivate” for activation.

“Set” commands can be issued to adjust the external parameters of objects, e.g. the location and size of a button, the resolution of a tracking area, etc. Once an object has been configured with “Set”, it can be started and stopped as needed with “Activate” and “Deactivate” commands. Once activated, visual interface widgets begin to monitor the video stream and return relevant events to the application.

The following XML string exemplifies a typical VIML-based command. It directs the VB-UI system to set the parameters of the VIconfiguration called “cfg” so that the boundaries of the internal coordinate frame are 500 units in x and y. It also sets the parameters of two widgets in the configuration, a button named “done”, which is located at x=200, y=200 and is 50 units large, and a track area which is 100 units in x and y and located at the origin (0,0) of the configuration coordinate frame.

```
<set id="uniqueID1001">
  <VIconfiguration name="cfg" left="0" right="0" top="500" bottom="500">
    <VIButton name="done" x="200" y="200" size="50" />
    <VItrackArea name="T1" left="0" right="0" top="50" bottom="50" />
  </VIconfiguration>
</set>
```

When a widget detects a user interaction, it returns a VIML event to the application. VIML events are XML valid strings that can be parsed by the application. These events are interpreted and handled by the application to control the flow of execution. The syntax of VIML events, as well as other objects and methods, is beyond the scope of this paper, and will be available soon in a publication format.

5 Example Application: A Multi-Surface Projected Store Index

One example of the experimental applications developed with this framework uses a device called an Everywhere Display projector (ED) to provide information access in retail spaces. This application provides a good example of how our dynamically reconfigurable vision system is used in practice.

5.1 The Everywhere Display

The ED is a device that combines steerable projector and camera, dynamic correction for oblique distortion, and a vision-based user interface system so it can direct a projected interactive interface onto virtually any planar surface. This allows visual information and interaction capabilities to be directed to a user when and where they are needed, without requiring the user to carry any device or for the physical environment to be wired. Figure 3 shows the current ED prototype (see [6] for details and other applications).

The current ED software consists of a three-tier architecture composed of a Services Layer, an Integration Layer and an Application Layer. The Services Layer contains the modules that control the projector, the gesture recognition system (the subject of this paper), and other system functions.

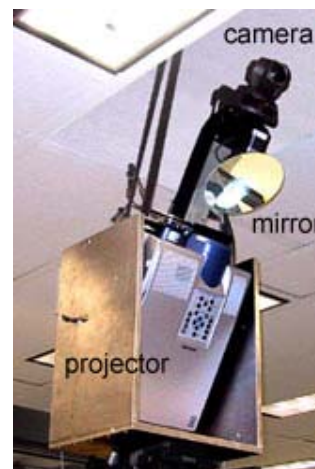


Fig. 3. The ED projector.

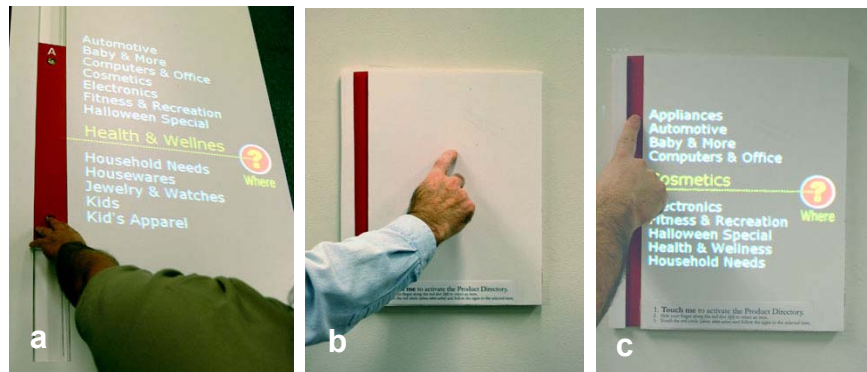


Fig. 4. The Product Finder application mapped onto different surfaces.

Each of the modules in the Services layer exposes a set of core capabilities through a specialized dialect of XML, e.g. VIML for the vision component. An ED interaction is accomplished by orchestrating the modules in the Service layer through a sequence of XML commands.

5.2 The Product Finder Application

The goal of this application is to allow a customer to look up products in a store directory, and then guide her to where the product is. This *Product Finder* is accessed in two forms. At the entrance to the store there is a table dedicated to this purpose, much like the directory often found at the entrance to a mall. Here there is a physical slider bar the user manipulates to navigate the projected index (see figure 4.a). Note that the slider has no physical sensors; its motion is detected by the vision system. Elsewhere in the store the Product Finder can be accessed using wall signs that look like the table at the entrance, with a red stripe on the left instead of a moving slider but with no image projected on them (figure 4.b). When a sign is touched (figure 4.b), the projector image is steered towards it, the store index is projected, and product search is accomplished in much the same way as on the table, except that moving the physical slider is replaced by the user sliding her finger on the red stripe (figure 4.c).

This application uses two vision interface configurations: a “call” configuration to request the Product Finder to be displayed on a particular surface; and a “selection” configuration to perform the product search. The “call” configuration consists of a touch button that covers the whole sign or table. The selection configuration consists of three widgets (figure 5). On the left of the configuration there are a widget designed to track the physical slider (red) and a widget designed to track the user’s fingertip (green). Only one of these will be active at a time. On the right is a touch button for the user to request directions to the selected item. The widgets are located with respect to the surface/configuration boundary area (the blue rectangle in figure 5). The corners of this area correspond to the corners of the wall signs.

In the current system, a single pan/tilt camera monitors the call surfaces using the information from a person tracking system. It automatically aims the camera to the sign or table nearest to the user (the current prototype is setup for a single shopper at a

time). Then, it activates the “call” configuration on that sign’s surface. In this way the system is always ready for the user to “call” the Product Finder.

When the user touches a sign or table, the “call button” widget sends an event to the application, which then projects the “selection” graphics on the sign, while activating the corresponding configuration on the sign’s surface. If the Product Finder is being

displayed on the table, the Physical Slider Tracker widget is activated and the Fingertip Tracker widget deactivated. On the wall signs, the reverse is true.

At this point the Product Finder is ready for use. The tracking widget sends events back to the application whenever the shopper moves their finger on the red stripe (or moves the slider), and the application modifies the display showing the product she has selected. When the user touches the “request directions” button, the application projects arrows on hanging boards that guide the shopper through the store, while the camera/vision system returns to monitoring signs.

This example demonstrates how the vision system can be easily switched between different configurations, how configurations are used on different surfaces, and how configurations are dynamically modified (by activating and deactivating widgets) to adapt to different contexts. The vision system architecture makes adding additional wall signs as easy as hanging the sign and defining a new surface for it.

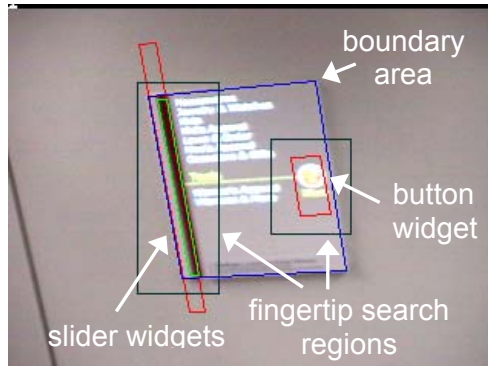


Fig. 5. Widgets mapped onto the surface of a wall sign.

6 Conclusion

The system described in this paper provides an application the ability to dynamically reconfigure a vision system that implements user interfaces. Interaction widgets recognize atomic interaction gestures, widgets are combined into configurations to support more complex interactions, and configurations are sequenced to create complete applications. These vision-based interactions can be created on the fly by an application that has little or no knowledge of computer vision, and then placed onto any calibrated planar surface in the environment.

The underlying architecture of the system, consisting of a dynamic tree of image processing components, combines flexibility, efficiency (through shared use of computational results), and easy code reuse and upgrading. Different applications and interactions can reuse the parameters of any given surface, which needs to be calibrated only once. Each widget can be tuned for best performance by parameters saved locally for each configuration and surface. The result is an “input device” that

can be dynamically configured by an application to support a wide range of novel interaction styles.

Our work in using a XML protocol for describing dynamically reconfigurable VB-UIs is part of a larger effort to develop similar protocols for the communication between an application and input/output service modules. So far we have also defined a protocol for controlling camera positions and movements (CAML) and another for the description and control of assemblies of projected images and video streams (PJML). In both cases a similar philosophy was applied, that is, decoupling of What, Where, and When; and run-time system structures that assemble input/output components on the fly. The system has been deployed in our laboratory and in two university laboratories where it is being used by graduate students and researchers.

References

1. Wu, Y. and Huang, T., *Vision-Based Gesture Recognition: A Review*. Lecture Notes in Artificial Intelligence **1739**, 1999.
2. *Proc. of the 5th International Conference on Automatic Face and Gesture Recognition (FG 2002)*. 2002, IEEE Computer Society: Washington, DC.
3. Turk, M., ed. *Proc. of the Workshop on Perceptual/Perceptive User Interfaces*. 2001: Orlando, Florida.
4. Kjeldsen, F., *Visual Recognition of Hand Gesture as a Practical Interface Modality*. 1997, Columbia University: New York, New York.
5. Pinhanez, C.S. and Bobick, A.F., *"It!": A Theater Play Featuring an Autonomous Computer Character*. To appear in *Presence: Teleoperators and Virtual Environments*, 2002.
6. Pinhanez, C. *The Everywhere Displays Projector: A Device to Create Ubiquitous Graphical Interfaces*. in *Proc. of Ubiquitous Computing 2001 (UbiComp'01)*. 2001. Atlanta, Georgia.
7. Zhang, Z., et al. *Visual Panel: Virtual Mouse, Keyboard, and 3D Controller with an Ordinary Piece of Paper*. in *Proc. ACM Perceptual/Perceptive User Interfaces Workshop (PUI'01)*, 2001. Florida, USA.
8. Pingali, G., et al. *User-Following Displays*. in *Proc. of the IEEE International Conference on Multimedia and Expo 2002 (ICME'02)*. 2002. Lausanne, Switzerland.
9. Starner, T., et al., *Augmented Reality through Wearable Computing*. *Presence* **6**(4), 1997: p. 386-398.
10. Kjeldsen, F. and Hartman, J. *Design Issues for Vision-based Computer Interaction Systems*. in *Proc. of the Workshop on Perceptual User Interfaces*. 2001. Orlando, Florida.
11. Kjeldsen, F., et al. *Interacting with Steerable Projected Displays*. in *Proc. of the 5th International Conference on Automatic Face and Gesture Recognition (FG'02)*. 2002. Washington, DC.