Machine Vision and Applications (2004) 16: 6–12
Digital Object Identifier (DOI) 10.1007/s00138-004-0145-6

Machine Vision and Applications

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, USA
(e-mail: {fcmk,levas,pinhanez}@us.ibm.com)

Published online: 13 July 2004 –

Abstract. We describe a system that supports practical, vision-based user interfaces, addressing the issues of a usable interaction paradigm, support for application developers, and support for application deployment in real-world environments. Interfaces are defined as configurations of predefined interactive widgets that can be moved from one surface to another. Complex interfaces can be dynamically reconfigured, changing both form and location on the fly, because the functional definition of the interface is decoupled from the specification of its location in the environment. We illustrate the power of such an architecture in the context of projected interactive displays.

Keywords: Vision-based interaction – Perceptual user interfaces

1 Introduction

Vision-based user interfaces (VB–UI) are an emerging area of user interface technology where the user's intentional gestures are detected visually, interpreted, and used to control an application. Although the recognition of human gesture and action has been the topic of many workshops and conferences [13,11,1] and the focus of much of our previous work [3,8], the issues surrounding making such systems practical and efficient have received far less attention. In this paper we examine the important characteristics in a practical VB–UI system and describe a system architecture that addresses some of the issues involved.

To be practical, a VB–UI system must meet the needs of both end users and application builders. For the user, the interface must be powerful enough to perform his task, intuitive enough that it does not distract, and efficient enough to leave machine resources available for other applications. For the application builder the system must be flexible enough to support a range of applications, and interfaces must be easy to create. To support deployment the system must easily adapt to different environments. In most current vision interface systems many of these issues have not been addressed. The visual recognition often performs well, but the system as a whole is hard-coded to perform a fixed task under limited circumstances. The remainder of this section will discuss in more detail these issues and how we address them.

As a user interface device, vision systems can be used in two distinct ways. One is to detect a user's "natural" state and behavior in an effort to make the computer respond to the user's needs more effectively. The user should not have to do anything outside his normal routine. We refer to these as "implicit" interactions. This paper addresses the other approach, "explicit" interactions, where there are specific commands in some gestural language that the user can perform to control an application.

Explicit control languages present two interesting problems. First, in the absence of some memory aid, even moderately complex gesture interfaces can be difficult to both learn and remember [3]. Second, the user can be subjected to the "Midas Touch problem" where incidental gestures may be misinterpreted as commands. Put another way, the system needs some way of knowing when to attend to user actions.

We can address both these problems at once by associating gestural actions with a *target*, meaning some visible entity in the environment where, and in relation to which, gestures are performed. A target may be an icon projected by the system, some physical object, or a distinctive location that is made "responsive" by the vision system [2]. A target provides both a memory aid and a location where the interaction is to be performed. This helps constrain the visual recognition problem, reducing both the computational load and the chance of false positive responses.

Up till now most vision-based interfaces have been built by vision experts, but as we go forward it is likely that visual gesture recognition will need to become a service that is easily used by application designers and programmers having little knowledge of computer vision. To address that issue, we provide a consistent and extensible high level of abstraction for the application programmer to use in designing vision-based interfaces. To create an interface, an application builds a *configuration* of *widgets* (describing *what* the interface is). Each widget responds to gestures in its vicinity, and the widgets are generally positioned so that they overlay some visible object

in the environment, thus creating a target for an interaction. A similar idea, referred to as VIcons, was put forward in [14].

Based on the description of the configurations, the vision system assembles a set of image processing components that implement the interface. To change the interaction, a new configuration can be sent to the system at any time. When a widget detects a user interaction, it returns an event to the application. The simplicity and similarity of this design to existing user interface programming methods make it relatively simple for a nonvision expert to use.

An important issue in taking VB–UI beyond the lab is ease of deployment in different environments. Some environmental variation, such as lighting, requires careful design of the image processing algorithms. Other variation, such as camera geometry, can be addressed at the architectural level and abstracted so as to make the application designer's job as simple as possible.

Because we are using targets for our interactions, one problem that arises is the relative geometry of the camera, the user, and the targets. We provide for the deployment of an interface onto arbitrary planar surfaces viewed from arbitrary camera angles. The parameters of the surfaces where the interface can be realized are defined and stored independently of the definition of the interface itself.

By explicitly decoupling the information describing *what* capabilities an interface provides (its semantic components and spatial layout) from *where* it appears in an environment/camera image, we provide a straightforward level of abstraction for the interface designer while facilitating (1) the porting of an application to a new environment where the imaging geometry and interaction surfaces are different, (2) the use of one surface for multiple applications, and (3) the use of the same interface on multiple surfaces.

The decoupling of *what* and *where* helps vision-based applications adapt to different environments, but it is equally important when an interface can be steered around the environment dynamically to respond to the current situation [7] or to follow the user [6]. The framework presented in this paper is also appropriate for situations where the interface surface is not static, for instance in the cardboard interface described in [15], or when the camera moves with the user in augmented reality or wearable computing [9].

The main contribution of this paper is the system architecture for the support of these *dynamically reconfigurable vision-based user interfaces*, which efficiently address the needs of the user and the application programmer.

2 Basic elements of dynamically reconfigurable VB–UIs

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

2.1 Configurations and widgets

In our framework, a VB–UI 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 that of traditional window-based applications, which are defined as a set of dialog windows, each containing elements such as scroll bars, buttons, and menus. Each widget performs a specific task for the application, such as triggering an event or generating a parameter value. In the case of VB–UI, a widget responds to specific gestures performed in relationship to it, such as a touch. Some of our earlier work describes the individual widget types we use and how they are implemented [4,5]. 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.

2.2 Surfaces

An application needs to be able to define the spatial layout of widgets with respect to each other and the physical environment, as that information is relevant to the user experience. It should not be concerned with details of the recognition process, such as where these widgets lie in the video stream. To provide this abstraction, we use the concept of interaction *surfaces*. A surface is essentially the camera's view of a plane in 3D space. Applications refer to surfaces by name and are typically not concerned with finer-grained details.

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 and a tracking area being mapped onto different surfaces. The process of determining the homography and establishing other local surface parameters is described in Sect. 3.3.

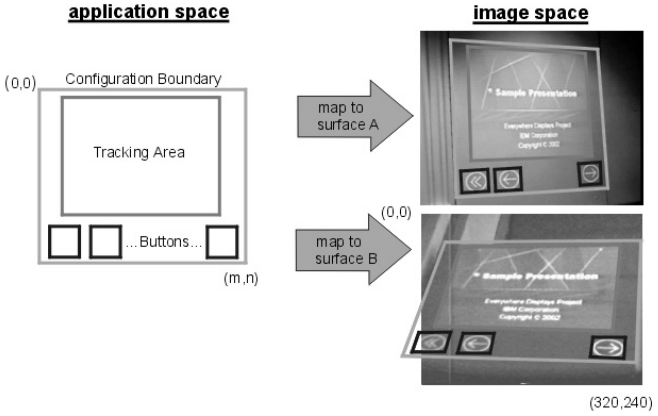

Fig. 1. Mapping a configuration onto two different surfaces

3 Architecture of a dynamically reconfigurable vision system

In order to efficiently support the 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 Fig. 2. There are components for image processing tasks such as finding the moving pixels in an image and tracking fingertips in the motion data, for analysis tasks such as looking for touchlike motions in the fingertip paths, and for system activities such as generating the events, storing widget parameters, and managing the surface homography.

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 Fig. 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 a widget designer has the option of specifying that these components be shared or not as needed.

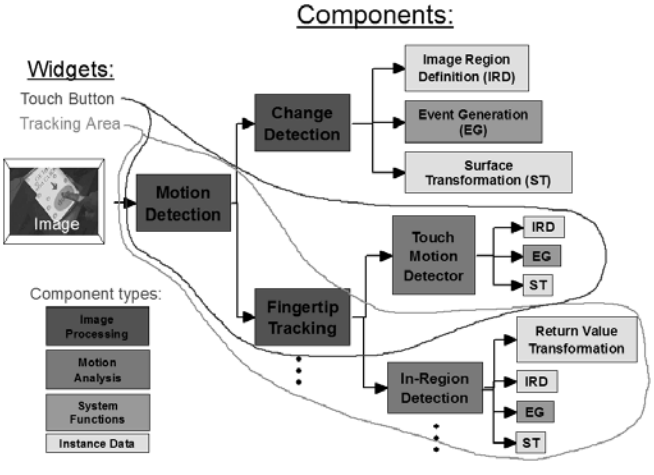

Fig. 2. Tree of components implementing three widgets

A good example of the tradeoffs of shared components is when using touch-sensitive buttons. If multiple buttons are active at one time, these buttons always 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 a telephone keypad. In other circumstances, such as when interacting with both hands, 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 flow 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 of, or in addition to, the original image.

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 four points in the image coordinate system that correspond to the corners of a configuration's boundary.

The image points can be located either manually or automatically and are then saved with the surface. Manual calibration consists of showing the user the video image on the screen and having the user identify the screen points corresponding to the corners of the configuration boundary. Automatic calibration consists of holding a rectangular patterned board on

the desired surface. The board is imaged by the camera to find the calibration points, which correspond to the corners of the configuration boundary. Finding the calibration points is straightforward and relies on the OpenCV routines FindChessBoardCornerGuesses and FindCornersSubPix. Unfortunately, due to the short baseline of practical calibration pattern boards, this technique is rarely accurate enough to use.

When a configuration is mapped to a surface, the corners of the configuration boundary and the surface calibration points form 4 point pairs, which are posted to the component tree. Each widget's Surface Transformation (ST) component then computes a homography that converts between the widget's configuration coordinates and image coordinates using the methods described in [12, pp 52–56). Later, during processing, components of the widget can query the ST to determine what image region to examine.

3.4 Image processing parameters

In order to get the best performance from the vision system, a number of parameters always seem to need adjusting. We keep these parameters hidden from the application so that the developer 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, widgets, and surfaces that have been defined, and parameters are maintained independently for each one. These parameters can be manually adjusted (and tested) from the vision system on-screen GUI.

Surface parameters include the location and distortion of the interface within the image and characteristics of the physical environment around that surface, such as the user's likely position while interacting with it.

Most of a widget's image processing components have several parameters that can be adjusted to tune performance. Because widget parameters are saved independently for each configuration, you can adjust each configuration to behave somewhat differently. 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.

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, with the spatial layout of the widgets specified in each. The sequence of configurations (as well as the non-UI aspects of the application) must be defined as a function of the events returned from the widgets combined with 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 and 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", which control when they are operational.

"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 that identifies the event type, configuration, and widget by name. 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 following XML string is a typical example of a VIML event sent by the vision system to the application:

```
<event id="002" >
  <VIconfiguration name="selector" >
     <VIbutton name="showWhere" >
        <VIeventTouch />
     </VIbutton>
  </VIconfiguration>
</event>.
```

The full syntax of VIML, which includes other objects and methods, is beyond the scope of this paper. The complete API will be formally published in the future, but a draft is currently available from the authors on request.

5 Example application: Projected information access in a retail environment

One example of the experimental applications developed with this framework uses a device called an Everywhere Display

projector to provide information access in retail spaces. This application provides a good example of how our dynamically reconfigurable vision system is used in practice. A full account of details of this application, as well as its design process, can be found in [10].

5.1 The Everywhere Display

The Everywhere Display (ED) (Fig. 3) is a device that combines a steerable projector/camera system, dynamic correction for oblique distortion, and the VB–UI system described here in order to project an interactive interface onto virtually any planar surface. Static cameras and software allow person tracking and geometric reasoning. ED 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 (see [7] for a more detailed description). Software assists with coordination of the various modules and provides an environment where complete applications can be built with comparative ease.

Fig. 3. Prototype ED projector

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 located. This *Product Finder* is accessed in two forms. At the entrance to the store is a table dedicated to this purpose, much like the directory often found at the entrance to a mall. Built into the table is a physical slider bar in a slot that

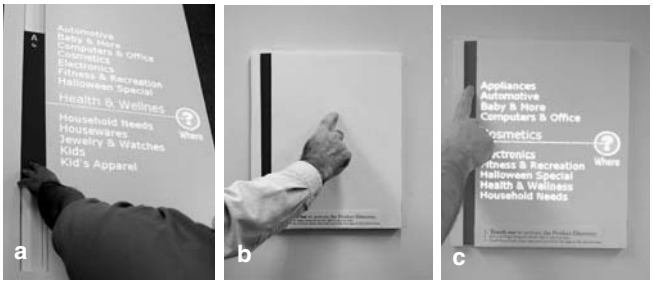

Fig. 4. The Product Finder application mapped onto different surfaces

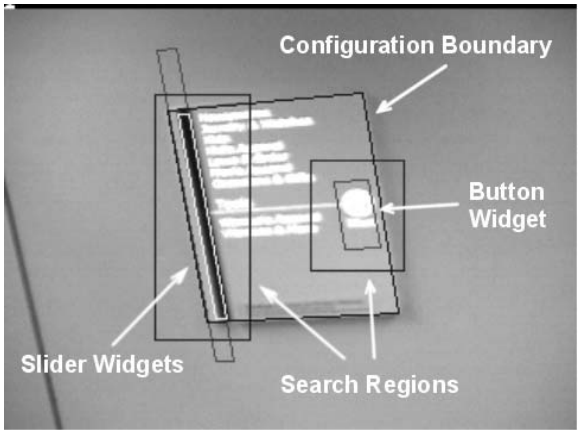

Fig. 5. Widgets mapped onto the surface of a wall sign

the user manipulates to scroll the projected index (Fig. 4a). Note that the slider has no physical sensors; its motion is detected by the vision system using the Physical Slider widget. 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. When a sign is touched (Fig. 4b), the projector image is steered towards it, the store index is projected, and the product search is accomplished in 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 (Fig. 4c). Again, there are no physical sensors on these signs.

This application uses two vision interface configurations: a "Call" configuration to request the Product Finder to be displayed on a particular surface and a "Select" 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 (Fig. 5). On the left of the configuration are both a widget designed to track the physical slider and a widget designed to track the user's fingertip. These are overlayed, and only one 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 configuration boundary. The corners of the boundary are mapped to the corners of the surface quadrilateral in the image. For the wall sign surfaces, these correspond to the corners of the wall signs themselves as shown in Fig. 5. For the table these correspond to marks on the slider table.

During operation ED is directed to track the customer, aiming the camera and placing the Call configuration on the nearest sign (or table) (the current prototype is set up for a single shopper at a time). When the user touches a call button, it sends an event to the application, which then projects the "Selection" graphics on the sign, while activating the Select configuration on the sign's surface. The Select Button widget, along with either the Physical Slider widget or the Fingertip Tracker widget, is activated as appropriate.

At this point the Product Finder is ready for use. The tracking widget sends events back to the application whenever the shopper moves her 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 Select button, the application projects arrows on various surfaces to guide

the shopper through the store, while the camera/vision system returns to monitoring signs.

This example demonstrates how an application can control a VB–UI by creating configurations with a known widget layout, directing them to different surfaces and modifying them dynamically (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.

5.3 The touch and track widgets

This section briefly describes the widgets used in the product finder application. For more details see [5].

Three types of widgets were needed to support the product finder, a touch button, a finger tracking slider, and a widget that tracked the physical slider bar.

Both sliders operate essentially the same way. They consist of a single image processing component that searches down from the top of the slider region till it finds an abrupt change in color value. They verify that the color transition is between the correct background and foreground colors (obtained by calibration marks on the table, and by using a skin color model on the other signs) and then generate a return value based on the location of the transition within the widget.

The major difference between the sliders is that the fingertip slider must take into account that the same hand will be used to select the value on the slider, then moved over to touch the Select button. Without some compensation, it was difficult for the user to select a value and have it remain unchanged when she moved to the Select button. This compensation is achieved by delaying reporting the transition location when it begins to drop till either it stops dropping or disappears. If the transition disappears, it is assumed the fingertip dropped down out of the widget as the user moved over to the button, and the drop in value is not reported. If it drops down then stops or comes back up, it is assumed that the user was just scrolling down, and the lower value is reported to the application. This simple heuristic does a very good job of keeping the correct value when the user withdraws her hand.

The touch button is somewhat more interesting than the sliders. Because the button is projected, it distorts the color of the fingertip inside it, making color analysis unreliable. The fingertip does change the apparent color of the projection, however, so the touch button first uses the Motion Detection component to create a mask image from the frame-to-frame changes in the video stream, morphological smoothing, and a vibration removal algorithm that ignores constantly changing pixels.

This motion mask is passed to the Fingertip Tracking component, which uses the prior knowledge of the user location to create a fingertip template of the correct orientation and size. This template is convolved over the motion mask in the vicinity of the button to locate fingertip hypotheses. The hypotheses are heuristically evaluated for match quality, distance from the user, etc. to choose the best candidate, which is tracked over time.

The recent path of the fingertip is passed to a Touch Detection component which looks for the characteristic out-pause-back motion of a touch, where the pause lies within the button.

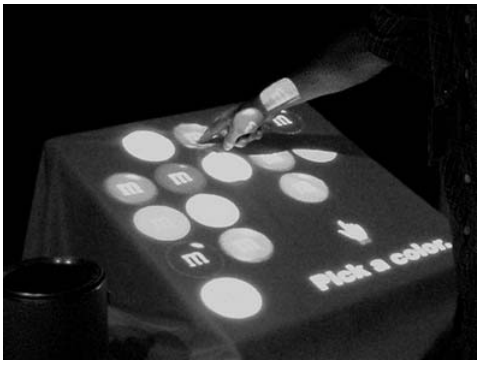

Fig. 6. A grid of interactive touch buttons

When it finds such a touch motion, it posts a request to the Event Generation component to send an event to the application.

This touch detection algorithm was tested using video of visitors to a demonstration of ED at SIGGRAPH '01. Six hundred twenty-one button touch events by 130 different users on 22 surfaces were evaluated. The best-performing surface, with a 94% success rate over 132 touches, was a grid of 18 buttons in close proximity (Fig. 6). The average success rate dropped to 81% by some surfaces that performed poorly due to poor image quality, frequent occlusion, and awkward ergonomics.

A recent reimplementation of the buttons based on a simpler and more robust algorithm shows better performance on the same data while reducing complexity, easing calibration, and reducing the need for parameter tuning. We will publish a detailed report on these new buttons in the near future.

6 Conclusion

The system described in this paper provides an application with the ability to create and dynamically reconfigure a vision-based user interface. Interaction widgets that recognize basic interaction gestures are combined into configurations to support more complex user dialogs. Configurations are used in sequence to create complete applications.

Configurations 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. Surfaces need to be calibrated only once and can then be reused by different applications and interactions. 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.

The system has been designed for a target-based interaction paradigm, where visible objects or projections in the environment are made interactive by placing widgets over them. This provides the user the advantage of a visible entity to help them remember the interaction and a spatial location for the interaction to help avoid inadvertently triggering an event.

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 modularity for easy code reuse and upgrading.

An XML protocol is used for communication between an application and the vision system.

We described a prototype application for customer navigation in a retail environment to demonstrate how the architecture can be used. We described the widgets used in this application and summarized the results of a study of their performance.

References

1. Davis L (ed) (2002) In: Proc. of the 5th International Conference on Automatic Face and Gesture Recognition (FG 2002), IEEE Computer Society, Washington DC
2. Ishii H, Ullmer B (1997) Tangible Bits: Towards Seamless Interfaces between People, Bits, and Atoms. In: Proc. of CHI'97, Atlanta, GA pp. 234–241
3. Kjeldsen F (1997) Visual Recognition of Hand Gesture as a Practical Interface Modality. PhD dissertation, Columbia University, New York, NY
4. Kjeldsen F, Hartman J (2001) Design Issues for Vision-based Computer Interaction Systems. In: Proc. of the Workshop on Perceptual User Interfaces. Orlando, FL
5. Kjeldsen F et al (2002) Interacting with Steerable Projected Displays. In: Proc. of the 5th International Conference on Automatic Face and Gesture Recognition (FG'02), Washington DC
6. Pingali G et al (2002) User-Following Displays. In: Proc. of the IEEE International Conference on Multimedia and Expo 2002 (ICME'02), Lausanne, Switzerland
7. Pinhanez C (2001) The Everywhere Displays Projector: A Device to Create Ubiquitous Graphical Interfaces. In: Proc. of Ubiquitous Computing 2001 (Ubicomp'01). Atlanta, GA
8. Pinhanez CS, Bobick AF (2002) "It/": A Theater Play Featuring an Autonomous Computer Character. In: Presence: Teleoperators and Virtual Environments 11(5):536–548
9. Starner T et al (1997) Augmented Reality through Wearable Computing. In: Presence: Teleoperators and Virtual Environments 6(4):386-398
10. Sukaviriya N, Podlaseck M, Kjeldsen R, Levas A, Pingali G, Pinhanez C (2003) Embedding Interactions in a retail Store Environment: The Design and Lessons Learned. In:Proc. of the 9th IFIP TC13 International Conference on Human-Computer Interaction (INTERACT03), September 2003
11. Turk M (ed) (2001) Proc. of the Workshop on Perceptual/Perceptive User Interfaces. Orlando, FL
12. Wolberg G (1994) Digital Image Warping, IEEE Press, New York, NY
13. Wu Y, Huang T (1999) Vision-Based Gesture Recognition: A Review. Lecture Notes in Artificial Intelligence, vol 1739. Springer, Berlin Heidelberg New York
14. Ye G, Corsco J, Burschka D, Hager G (2003), VICs: A Modular Vision-Based HCI Framework. In: Proc. of the 3rd International Conference on Vision Systems, Graz, Austria. pp 257–267
15. Zhang Z et al (2001) Visual Panel: Virtual Mouse, Keyboard, and 3D Controller with an Ordinary Piece of Paper. In: Proc. of the ACM Workshop on Perceptual/Perceptive User Interfaces (PUI'01), Orlando, FL