ISSN 2250-3153

Extensible GUIs for Remote Application Control on Mobile Devices

Ajitkaur Saini, Bhushan Borse[*], Pratiksha Kulkarni[**]

[*] Computer Department, R.A.I.T
[**] Electronics & Telecommunication Department, R.A.I.T

Abstract- A chasm exists between the desktop and mobile worlds in terms of hardware performance, graphics capabilities and input devices. Some kinds of software, such as 3D graphics or advanced word-processing tools, are still out of reach for handheld devices. So, although redesigning a desktop application for mobile environments is often possible, mobile devices' limitations discourage using such software in nomadic scenarios. One type of solution to this problem employs well-known remote-computing techniques in which a remote server executes specific application and sends only the resulting graphics representations to the client devices a sequence of still images or a video stream.1, 2 In most cases, these techniques require shrinking the server's video output (including the application interface) to fit mobile displays' low resolution and wireless-network bandwidth. This results in poor quality at the mobile site that severely impairs application usability. To overcome this limitation, a second type of solution employs ad hoc remote visualization to deliver only the application work area (the area displaying the effect of user interface manipulation) to the client side while redesigning the application interface from scratch. But this wastes time and money and results in a strict dependency on the original application. A third possible solution involves Web applications and, more generally, Web operating systems. However, development of enabling technologies in the mobile scenario is at an embryonic stage. So, only a few of the existing implementations can run on today's devices.

We've developed a fourth solution: a software independent approach that extends the basic remote-control paradigms to maintain a separate work area and interface. Using image processing, our approach automatically analyzes and classifies the server-based application and then generates an interface description. On the client side, the user's mobile application reloads the description. Any interaction with the client interface produces an interaction on the corresponding graphics element in the application interface on the server. This approach manages the work area as a flow of still pictures that the server delivers to the client. Prototype implementations show how our solution can effectively transfer simple and complex applications to mobile environments, letting users remotely interact with applications

Index Terms- GUI, Remote Application, Mobile Devices.

I. INTRODUCTION

Any strategy for designing mobile-device applications must clearly separate GUI elements from the work area. Then, we can adopt optimization (and automatic-generation) approaches to tailor layout, appearance, and interaction modalities to both user needs and device capabilities. To enable all this and allow deployment of existing desktop applications on mobile devices, we've developed a novel client-server framework as an evolution of established remote-computing and remote-visualization solutions.

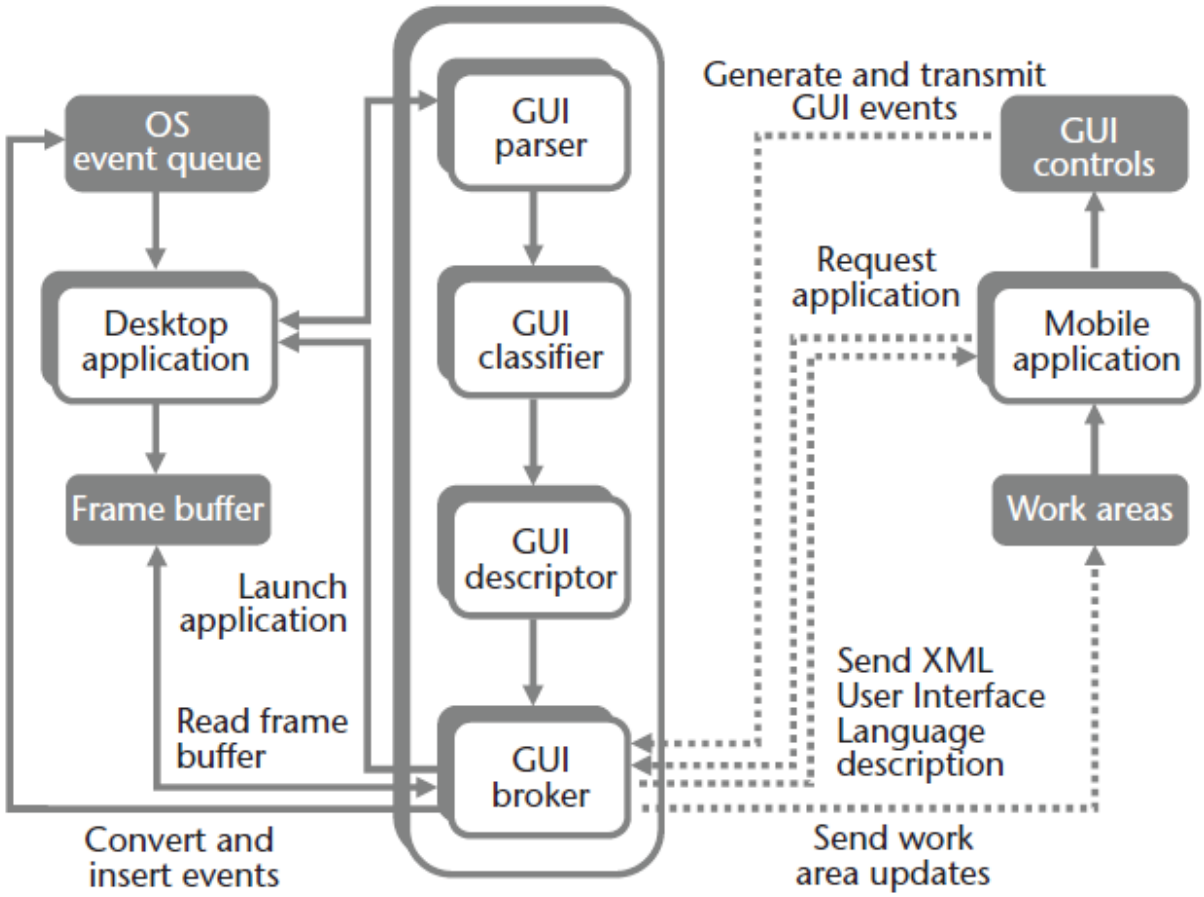


Figure 1. An overview of the architecture for extending GUIs to mobile devices.

The four main elements—the GUI parser, classifier, descriptor and broker modules—reside on the server.

Figure 1 shows the overall architecture. On the server side, the GUI parser analyzes the interface of an application running on the same machine by simulating mouse movements and clicks. It can identify the image elements belonging to the graphical lexicon that were used to build the application.

The GUI classifier then places each element in a specific category (for example, menus, buttons, check boxes, and work areas). The GUI classifier can also disclose element relations with the surrounding context. For example, it might classify an element as a combo box item because that element is in an image that appears only when the user presses the combo box's drop-down arrow.

When this classification is complete, the category, location, size and container are known for all GUI elements. The GUI descriptor converts this information into a User Interface Description Language (UIDL) file containing the whole interface's description. A mobile application connects to the GUI broker and requests access to an application. The GUI broker sends the UIDL description to the client and launches the requested application on the server.

The mobile application parses the description file and presents to the user all the constituent elements, which the user can rearrange to create a personalized GUI. The GUI broker converts user generated events into suitable mouse and keyboard events to insert into the OS event queue. Finally, the GUI broker transmits images representing work area updates to the mobile device, giving the user the impression of a local interaction.

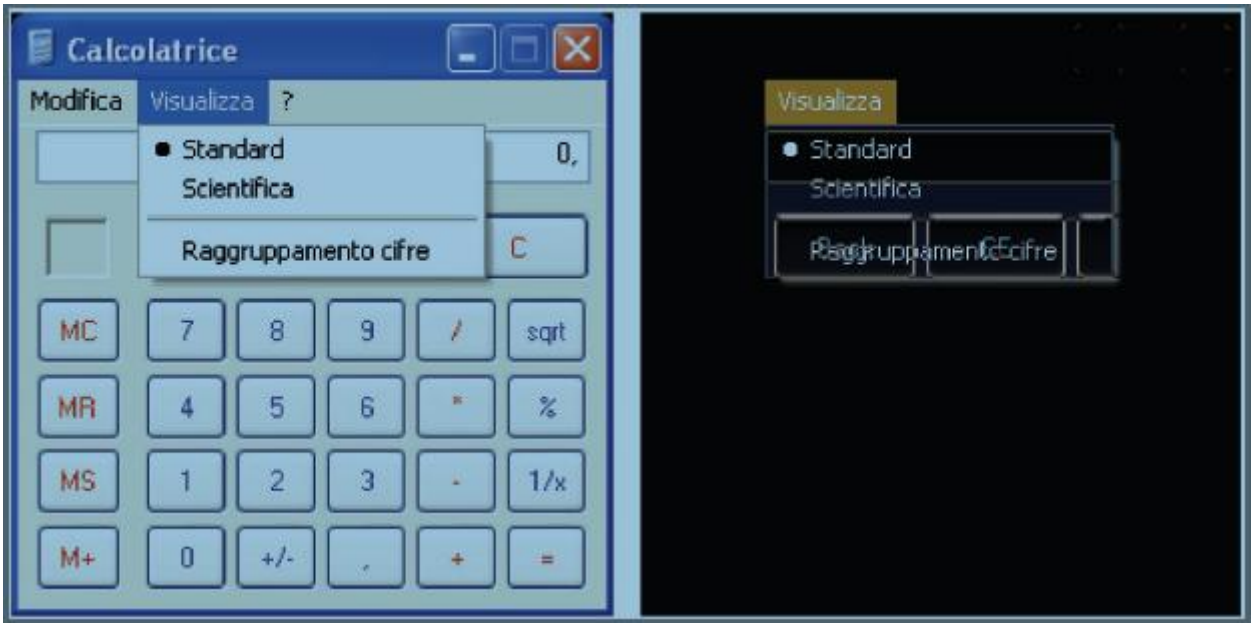
II. GUI ELEMENTS

Using trivial edge detection and pattern-matching techniques, we can discover a white background text area, a radio button group with characteristic circular icons, or a bounding box. However, analyzing the user interface to produce a functional description to reuse in application development takes more than just image processing. For more complex elements such as menus and combo boxes, we must investigate in depth their behavior with respect to user interaction. Moreover, a main goal of designing the GUI parser was to achieve a high portability across various platforms. This will help develop an infrastructure that can work with different OSs with different GUI styles.

One aspect common to almost all window based operating systems is highlighting. When the mouse moves over an interface element, this element is highlighted; that is, it becomes brighter, is bounded with a colored box, or becomes 3D. Independent of the specific highlighting technique, this effect helps focus user attention on that element. Even more important, thanks to this behavior, the system signals any user interface element's presence and position. This lets the user interface participate in element identification (and classification), thus easing the GUI parser's job. For example, clicking on a highlighted box

containing some text could open a drop-down list, signifying that this element is probably a menu.

So, for the GUI parser to locate GUI elements, we obtain from the frame buffer the graphical representation of the user interface before and after element highlighting. Then, we simply compute the difference (that is, the XOR) between them to get the element's position. The resulting image's pixels are all black except those corresponding to the element region, because the difference represents the highlighting itself.



(a) **(b)**

Figure 2. For the MS Windows Calculator
(a) Highlighting effect over a menu and
(b) Corresponding difference image. Computing the difference image lets us determine interface elements' location and size

A simple algorithm identifying the non-black region's boundaries gives the element's exact location and size (see Figure 2).This algorithm assumes that highlighted and non-highlighted representations of the user interface for a specific element are available. However, this requires prior knowledge of element position Moreover we need to instruct the system to take a snapshot of the user interface before and after positioning the mouse over the selected element.

Because we aimed to design an automatic tool for interface description, we had to integrate this algorithm into a programmable logic to produce highlighting when needed. We achieved this through an ad hoc algorithm that interacts with the OS's native event queue and mimics mouse movements and mouse clicks through specific system calls. This algorithm has two phases. The first identifies the interface's basic elements, including buttons; check boxes, text fields, radio buttons, and sliders. This phase moves the mouse using discrete steps from the interface's upper left corner to the bottom right corner over imaginary horizontal lines. We compute the difference images and identify the elements' location and size. Figure 3 shows this phase's results for the Blender (www.blender.org) 3D modeling tool.

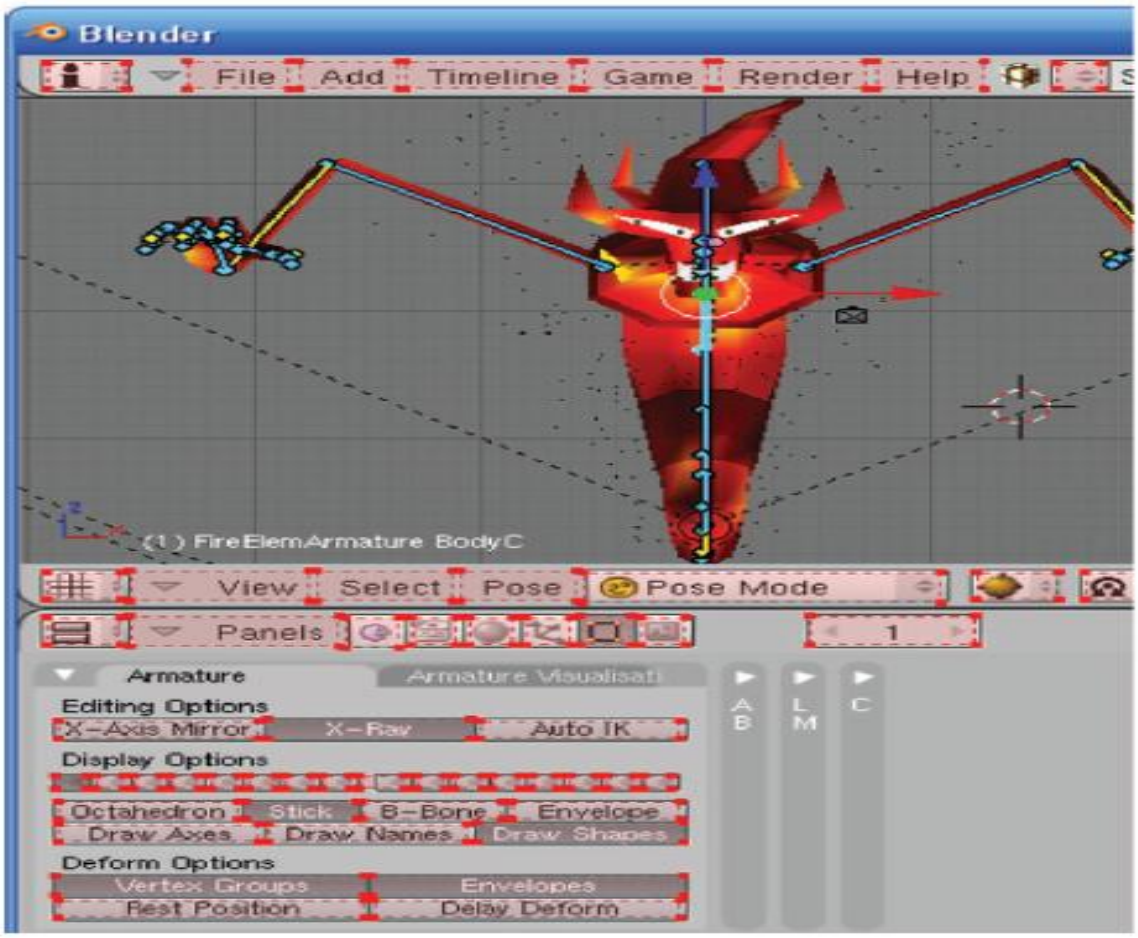


Figure 3.Results from the GUI parser for the Blender 3D modeling tool.

The parser has identified elements of the user interface

Some elements identified in the first phase undergo another processing. This second phase correctly identifies menus and combo boxes that don't appear until the mouse interacts with them. In this case, the logic moves the mouse over the various elements and simulates a mouse click, simultaneously collecting the highlighted representation. An ad hoc algorithm processes the difference image and retrieves required information including submenus, icons and hot keys.

To speed up the process, we created wizards that let users specify interface regions. With the wizards, users can also analyze and select the application work area. For example, in a common WYSIWYG (What You See Is What You Get) word-processing application such as Microsoft Word, the work area is the white page in which the user types. In a 3D modeler, it will probably be the various model views (that is, top, front, side, and camera).

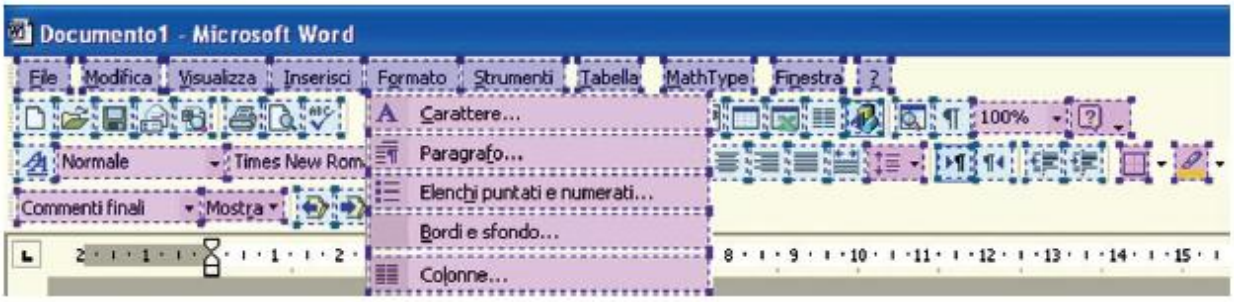


Figure 4. Results from the GUI classifier for MS Word.

Different colors indicate different categories of graphics elements that the classifier has identified.

III. Classifying GUI elements

Given the variety of styles and the existence of nonstandard graphics building blocks in existing GUIs, designing a classification algorithm using traditional pattern recognition techniques is extremely difficult. Image resolution and contrast, which in some cases could be very low, further complicate this task.

To address this problem, we developed two ad hoc algorithms. The first segments the image representing a GUI element into its sub blocks. It analyzes the image from left to right; extracts information concerning each sub blocks width and content, and stores that information for later use. Optical-character-recognition (OCR) techniques recognize text in the element's bounding box.

The second algorithm works on the sub blocks, exploiting pattern-matching techniques (specific for each GUI element) to check for characteristic shapes and attributes. Then it assigns the element to a category. The parameters that control algorithm behavior include the number of sub blocks, existence of icons, distance between sub blocks, presence of characteristic shapes (for example, arrows), and number of text characters. Figure 4 shows the results for MS Word.

We designed this process to be as general as possible to adapt to various user interface styles. However, the pattern-recognition-based techniques only demonstrate the overall architecture's feasibility and aren't comprehensive. Nevertheless, as the following sections show, our current method works well enough to let us reuse complex existing applications on mobile devices.

IV. Generating the GUI description

To generate a reusable description of the interface, the GUI descriptor uses XUL (XML User Interface

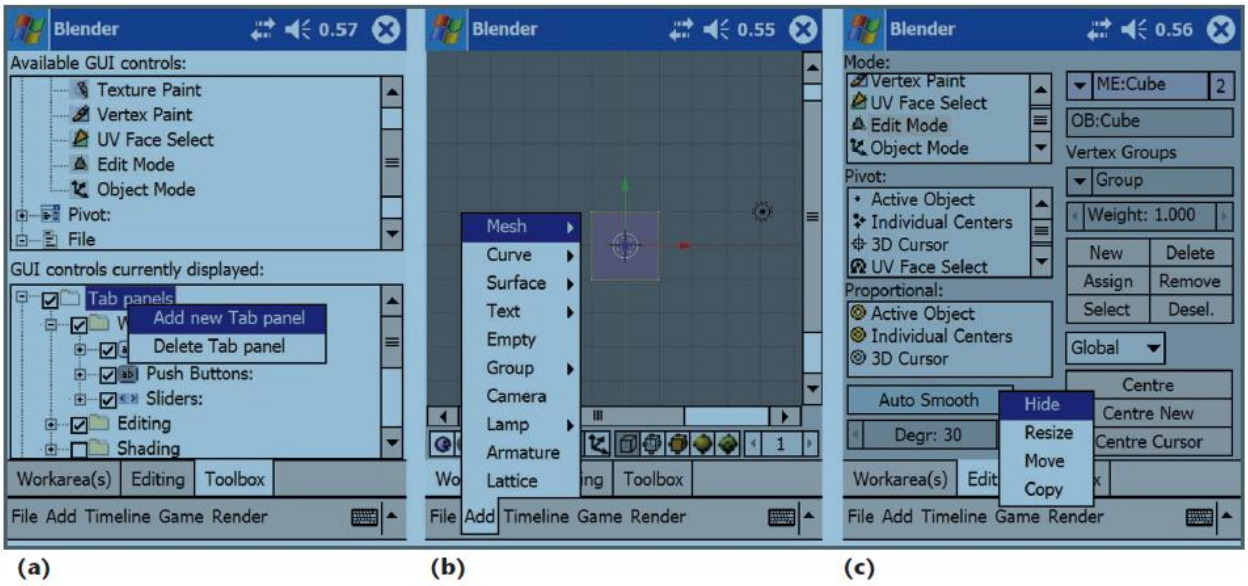


Figure 5. The Pocket PC remote-control application on a Dell Axim x50v PDA:
(a)A hierarchical tree view keeps track of the native GUI structure.
(b) Elements from the XUL description of the Blender interface have been placed into a tab panel.
(c) This display shows other elements of possible interest to a user.

Language). With XUL, the programmer specifies what the interface must include, not how it must be displayed. This separates the application's look and feel from its control logic. A predefined set of structured basic elements characterized by a predefined set of attributes extended by the user describes the interface. Basic elements can be nested to create complex controls found in GUIs. The main XUL elements allow the description of windows, buttons, menus, combo boxes, scroll bars, checkboxes, radio buttons, text fields, labels, and images (the complete XUL documentation is at www.mozilla.org/projects/xul).Despite XUL's expressiveness, it can't natively record the necessary information related to remote visualization and control. So, no XUL tag exists for describing the work area or the address of the server running the GUI broker. Moreover, XUL doesn't provide specific tags for all the elements used in modern interfaces. To overcome these limitations, we extended XUL. Specifically; we added a server tag to record the connection parameters for configuring the remote control session. We also defined work area and slider graphics tags. Finally, we inserted an attribute in the button tag to distinguish between normal buttons and push buttons.

V. Generating the GUI and providing remote interaction

The GUI descriptor saves the generated GUI description into an XUL file. Remote users can access the file when they need to use a portable version of a desktop application. For this, we've developed a remote-control application for a Pocket PC PDA, using Microsoft C#.net, and for Java-enabled mobile phones,

using J2ME (Java 2 Platform, Micro Edition). When the user connects to the server, the GUI broker sends the user a list of applications installed on the server. The user selects an application, and the GUI broker transmits the associated XUL description. (The system can be configured to activate the GUI identification, classification, and description tasks if the description isn't available.)

In the Pocket PC version, the mobile application displays a box containing the elements available for GUI construction, using a hierarchical tree view that keeps track of the native GUI structure (see Figure 5a). To customize the GUI, the user can drag and drop icons between this tree and an underlying tree displaying the elements to be placed in the interface. The user can arrange elements into tab panels, menus, and context menus (which appear when the user presses the stylus over the touch screen for a while). In a tab panel, the user can reorganize elements as needed, even radically changing the interface's original appearance. So, elements originally organized through a menu can be positioned into a list box, and a combo box can display menu elements in the remote application's GUI. This lets the user find a more comfortable organization of GUI elements, allowing optimized application control even on platforms with limited input devices.

In Figure 5b, some GUI elements extracted from the XUL description of the Blender 3D modeler tool GUI have been placed into a tab panel. The panel's upper part hosts the work area. Below the work area, some Blender buttons have been positioned with a slider. Instead of relying on the native set of graphics controls made available by C#.net for the Pocket PC, we've created new GUI elements that start from basic building blocks such as panels and images. This lets the mobile application recreate the XUL description's content. Figure 5b shows how we've recreated original menus starting from information extracted by the OCR software. Figure 5c shows other GUI elements of possible interest to a user of this mobile application.

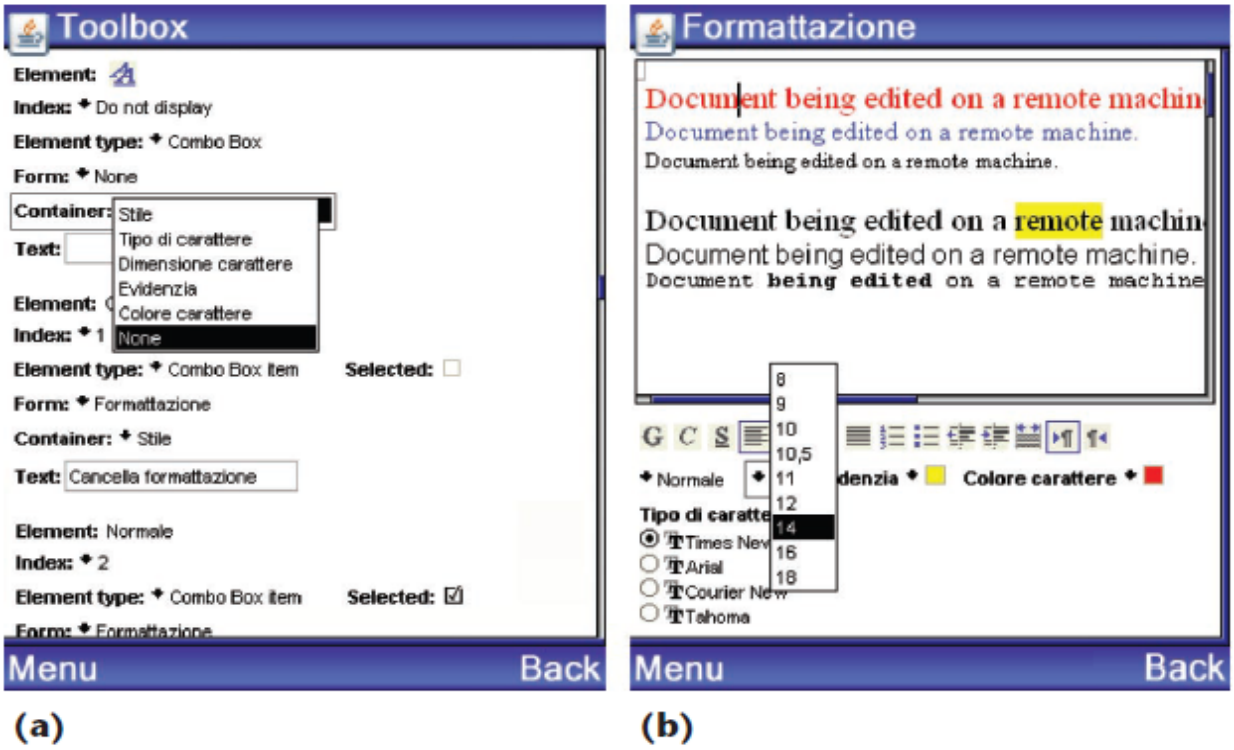


Figure 6. The Java J2ME remote-control application on a Nokia N80mobile phone:
(a) With this form, users can customize the MS Word GUI.
(b) This window displays a possible customized interface for remote control.

Here we can see how, on the basis of user preferences, our framework has translated the original combo boxes into list boxes to let the user display both the original icon and text. Native C#.net controls wouldn't have allowed this. By using a context menu, the user can hide each GUI element, resize it, or move it within the current tab panel or to another one. This lets the user precisely control any aspect related to the interface's final appearance and behavior.

The Java J2ME version provides comparable functionalities. However, we had to adjust its appearance and control capabilities to account for the limitations of a mobile phone's screen size and input devices.

Figure 6(a) shows the initial form allowing GUI customization for MS Word over a Nokia N80 phone. Figure 6(b) shows a possible customized interface for remote control. When the user manipulates GUI controls (for example, presses a button, selects a menu item, or activates a check box) or interacts with the work area by moving and tapping the PDA stylus or using the phone keyboard, the device transmits the corresponding events to the GUI broker. On the basis of the knowledge of

element location in the original GUI, the GUI broker converts the received information into suitable mouse and keyboard events and inserts them into the OS event queue. When the system processes these events, they affect the work area's appearance. The GUI broker extracts the updated representation of the work area from the frame buffer and sends it to the mobile application, where it's redrawn on the device display.

Currently, we encode work area updates as JPEG images. This encoding strategy can be replaced with any other solution commonly used in remote visualization architectures2 without significantly altering our approach's overall philosophy. Doing this might achieve higher update frame rates and lower latencies.
This solution has two main advantages over similar techniques (see the sidebar for a discussion of some techniques). First, remote applications don't require modification. Graphic elements of the GUI at the client site are directly connected to the corresponding objects at the server site. Second, the user can customize the mobile device's GUI, thus fulfilling most of the guidelines for (handheld) mobile-device interface design.3 According to these guidelines, GUI elements for mobile devices should be as similar as possible to the corresponding items of a desktop interface to preserve consistency (or continuity). Our approach lets the mobile device's GUI recreate the desktop application's original icon and text for buttons, check boxes, menus, and so on. This maintains the native application look and feel. Moreover, the user can personalize the element placement and GUI appearance. Original elements can be radically transformed (menus can become list boxes, buttons can be grouped into combo boxes, and soon) or discarded. This enhances GUI aesthetics and improves the mobile application's attractiveness and usability.

Our solution is based on the remote-visualization paradigm. So, from the client GUI viewpoint, memory load isn't a concern because it mainly affects the application on the server while preserving local-resource usage. The interface's configurability lets the user manage shortcuts and other items enabling access to special functions. Moreover, this approach lets users implement a hierarchical organization on the mobile device even when such organization is missing (or limited) in the desktop application. Furthermore, they can integrate access to hidden application logic, multimedia and multimodal interaction, and context-awareness functionalities in the mobile GUI without redesigning or rewriting the original applications.

This solution's limitations mainly concern the detection and classification of graphics elements that produce dynamic changes over the GUI. A pressed button can change both the work area's content and the interface itself (that is, create new graphic elements such as panels and dialog boxes).In other cases, GUI manipulation can cause modifications that affect previously existing elements. We've partially solved these problems by letting the user intervene during GUI analysis through the ad hoc wizards. At this stage, our approach can't manage these situations automatically. So, future research will aim to integrate techniques for continuously processing the remote-application GUI, detecting dynamic changes, and generating on-the-fly descriptions of the interface for the client device.

REFERENCES

[1] "RealityServer Functional Overview," white paper, Mental Images 2007, www.mentalimages.com/2_3_realityserver/RealityServer_Functional_Overview.pdf.

[2] S. Stegmaier et al., "Widening the Remote Visualization Bottleneck," Proc. 3rd Int'l Symp. Image and Signal Processing and Analysis, IEEE Press, 2003, pp. 174–179.

[3] J. Gong and P. Tarasewich, "Guidelines for Handheld Mobile Device Interface Design," Proc. Decision Sciences Inst., Decision Sciences Inst., 2004

AUTHORS

First Author – Ajitkaur Saini, Bachelors in Computer Engineering, R.A.I.T, rampe17292@gmail.com
Second Author – Bhushan Borse, Bachelors in Computer Engineering, R.A.I.T,bhushanyborse@gmail.com
Third Author – Pratiksha Kulkarni, Bachelors in Electronics and Telecommunication, R.A.I.T, pratiksha018@gmail.com

Correspondence Author – Pratiksha Kulkarni, pratiksha018@gmail.com, (+91) 98 92 336 913.