Running Graphical Desktop Applications on Tactile Graphics Displays Made Easy

Christiane Taras, Michael Raschke, Thomas Schlegel, and Thomas Ertl
Institute of Visualization and Interactive Systems, University of Stuttgart,
Universitaetsstr. 38, 70569 Stuttgart, Germany
Email: taras, raschkml, schlegel, ertl@vis.uni-stuttgart.de

***Abstract*—Tactile graphics displays have improved over the years and their importance for blind people and their education has been shown. Unfortunately, most of these devices can be used only with applications specifically developed for them. In this paper, we present a new framework for tactile rendering which makes it easy to run graphical desktop applications on a tactile display. Our framework utilizes a filter program to analyse the application's user interface without having access to the application's source code. The output is converted to a tactile user interface with the help of configurable rendering methods. As the rendering is based on widgets that can adapt their layout to the target output size, the tactile rendering can easily be adapted to devices of different sizes. We present the concept of our framework and demonstrate how it can be applied to existing applications using Windows Solitaire as an example. The framework was developed within the HyperBraille project, but is encapsulated into independent components that can be easily be reused in other projects. We believe that our framework will ease promoting the development and spreading of tactile graphics displays.**

I. Introduction

Graphical information plays an important role in our daily life. Whether it is a diagram in information materials, a graph of a mathematical function, a part of a graphical user interface or something like a tile in a web-based scrabble game - graphics are everywhere. For visually enabled people, graphics support understanding and give joy to the life. The same is for blind and visually disabled as for example Aldrich and Sheppard [12], [24] showed. For that reason, a lot of research is done in the field of preparing tactile graphics for school books, maps and many more (see for example [4], [5], [6], [7], [8] and [18]).

Due to the complexity of most graphics, interactive exploration of tactile graphics is much more useful than working with static ones alone. Therefore, for several years different groups of researchers have been working on the development and improvement of tactile graphics displays. Vidal-Verdú and Hafez [25] gave a good overview on previous work. Watanabe, Kobayashi et al. showed that tactile graphics displays can heavily support the communication between teachers and blind children [27] and between blind persons when talking about graphical information [15]. Furthermore, tactile graphics displays reveal the possibility of rapidly changing the presented graphic which helps in keeping the graphics up-to-date. So it's worth performing research in the field of tactile graphics displays.

Unfortunately, this field has two major problems. The first is, that the displays themselves are quite expensive or not at the market at all. The second is, that there is not enough useful software that drives these displays. Until now, only some special applications like [14], [21] and [22] were developed for tactile graphics displays. The framework that we present in this paper is meant to contribute in solving the second problem. It does not provide guidelines how to design an optimal tactile user interface. But it provides a technical means to easily convert ordinary graphical user interfaces into tactile representation. Thereby, different possibilities to optimize the rendering according to users preferences or characteristics of special applications were implemented. Only if a large number of ordinary desktop applications can be run on tactile graphics displays easily those devices will be accepted as valuable working equipment for visually impaired people. This would raise market demand and thereby through higher production output the prices of such displays probably would decrease. So, besides improving the displays and their production processes themselves, providing frameworks for valuable software is a very important aspect to promote tactile graphics displays.

The framework was developed as part of the HyperBraille project [1]. The goal of this project is to develop a touch-sensitive tactile graphics display combined with a graphical screen reader to better support blind and visually impaired people in controlling graphical user interfaces. The current prototype of the display is the BrailleDis 9000 [26] which was enhanced during the project. The BrailleDis 9000 provides 120×60 pins that can either be lifted or lowered. It's surface is touch-sensitive. So, performing input via gestures on the display is possible.

II. Presenting GUIs to Blind Users

Currently, blind people mostly use screen readers to access graphical user interfaces. The screen reader analyses the user interface and gives information about it via speech or Braille output. Soon it was recognized, that not only real text is important for the user but also information about other interface elements has to be presented. So, for example, also the type and state of an interface element is spoken or presented in Braille. For the Braille output it is also possible to present simple layout information. Thereby, the user can, for example, find out if an element is aligned to the left of a dialog or if a number of elements are aligned in one line. As Braille displays

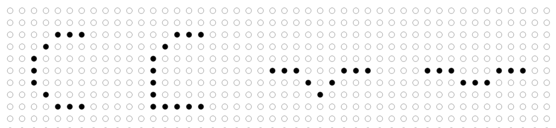


Fig. 1. Some examples for widget modules. From left to right: marker for elements that can be pressed, marker for editable elements, marker for elements that can be opened, marker for an opened element

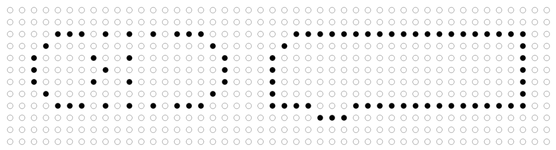


Fig. 2. Two examples of widgets, that are composed from widget modules, differently styled borders and Braille text. Left: a disabled "OK" button. Right: an enabled edit field that can be opened (i.e. a combobox with an input field).

can only present a small number of signs the type information is often shortened. For example, the buttons of the message box "Do you want to save the changes?" are presented as "btn Yes btn No btn Cancel". The active button is often underlined or marked with a blinking cursor. But, all in all, the information stays textual. The user has to distinguish if the presented information originates from real text or describes a user interface element.

For tactile graphics displays one can go a step further. There, one is not bound to textual information but can also use borders to surround text or group elements or use small graphics that can even exceed the bounds of ordinary Braille signs. So a kind of tactile widgets can be created that combine graphical elements with Braille text and thereby ease accessing graphical user interfaces with tactile displays. Graphical elements can represent types, states and grouping of interface components in a much more compact and flexible way than text alone can do. For this reason, in the context of the HyperBraille project, Kristin Albert designed a set of tactile widgets [11] to represent ordinary graphical user interfaces. Therefore, she analysed visual widgets for their interaction possibilities. From this she created a modular widget set with a small number of tactile modules which represent the interaction that is possible for a widget (see 1). Those modules are combined with different border patterns to compose the tactile widgets (see 2).

Albert's widgets were utilized in different user studies during the HyperBraille project (e.g. [17]). First user feedback has shown, that tactile widgets are a useful means for presenting graphical user interfaces on tactile graphics displays. Of course, the widgets are further investigated and improved. For example Denise Prescher evaluated a set of scrollbars for a tactile windowing system [20]. Furthermore, Albert's widgets were partly derived from TAWIS, the graphical screen reader of Friedrich Lühti. A blind man who has long time experiences with working on a tactile graphics display. Unfortunately, his work is unpublished.

To further promote the tactile widgets and so establish a common "Look&Feel" for tactile user interfaces, we also use them in our tactile rendering framework for the default rendering. For the composition of a whole tactile user interface out of those widgets, we try to keep the original layout. By this, on the one hand, we keep groupings that are expressed only through positioning of widgets and not logically in the user interfaces widget hierarchy. On the other hand, keeping the original layout supports communication between users of the graphical interface and users of the tactile interface. User studies during the HyperBraille project have shown that this is a useful approach (see for example [23]).

III. Our Tactile Rendering Framework

As figure 3 shows, our tactile rendering framework supports the whole process necessary to run a graphical desktop application on a tactile graphics display. This includes configurable tactile rendering, as well as interaction handling. Thereby, it is neither bound to a certain application nor to a special display.

A. The Rendering Process

To apply our framework to a graphical desktop application only its executable is needed. This is plugged to the framework using an application filter. The filter analyses the application's user interface at runtime and provides the rendering framework with an object-based representation of the interface elements that have to be rendered. This includes information about the hierarchical composition, as well as attributes like type and name of the user interface elements. Using an object-base representation instead of just the graphical output of the application facilitates redesigning the user interface for the tactile representation. Different filters can be connected to the renderer to support various applications. In HyperBraille we have developed a couple of them for different applications like Microsoft PowerPoint, Microsoft Internet Explorer, Mozilla Firefox and ordinary Windows applications that support UIAutomation [9]. The filter also has to track changes in the application's user interface and inform the rendering framework about this. Figures 4 and 5 illustrate the output of a UIAutomation filter for an ordinary dialog.

The filter's output is converted into a tactile user interface via exchangeable rendering methods. The tactile user interface itself is built up with user interface components as known from graphical user interfaces for sighted people. In short, building the tactile output with our framework is like composing an ordinary GUI. By this rendering rules which were defined by other developers can easily be influenced. If the rendering methods only provided bitmaps, it would for example be hard to change the cell spacing of a table. Using widgets this is just a matter of changing the appropriate attribute. Furthermore, the tactile output can easily be adapted to differently sized displays by providing appropriate layout algorithms.

When the composition of the tactile user interface is finished, it is provided to the tactile graphics displays as bitmap. Different displays can be connected to our framework and even be driven synchronously at the same time. The output size can be configured according to the connected displays.

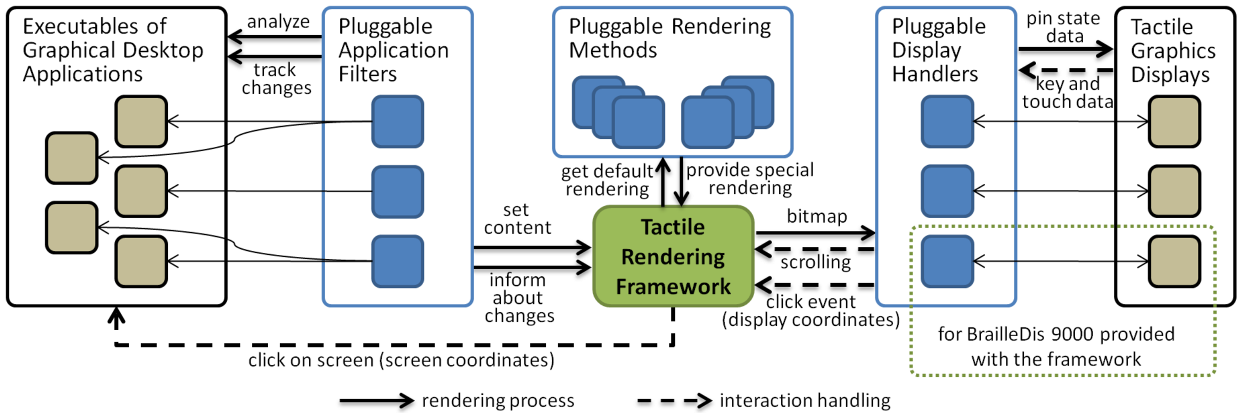
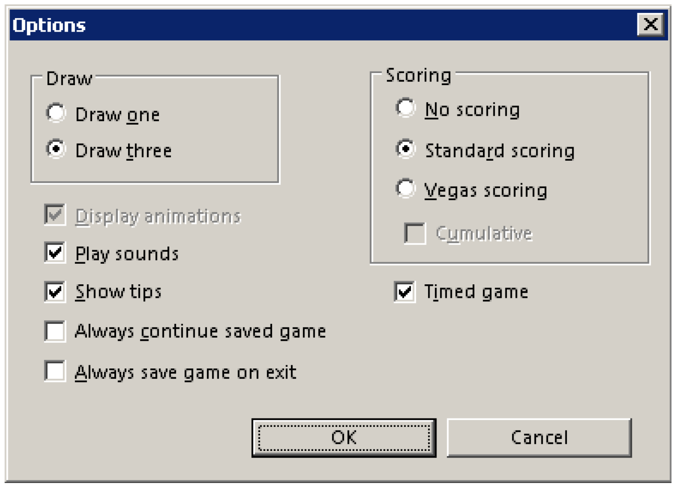

Executables of Graphical Desktop Applications
analyze
track changes
Pluggable Application Filters
Pluggable Rendering Methods
Pluggable Display Handlers
pin state data
key and touch data
Tactile Graphics Displays
get default rendering
provide special rendering
set content
inform about changes
Tactile Rendering Framework
bitmap
scrolling
click event (display coordinates)
for BrailleDis 9000 provided with the framework
click on screen (screen coordinates)
rendering process
interaction handling

Fig. 3. The process of rendering and click handling that is supported by the framework.



Options

Draw
- ◯ Draw one
- ● Draw three

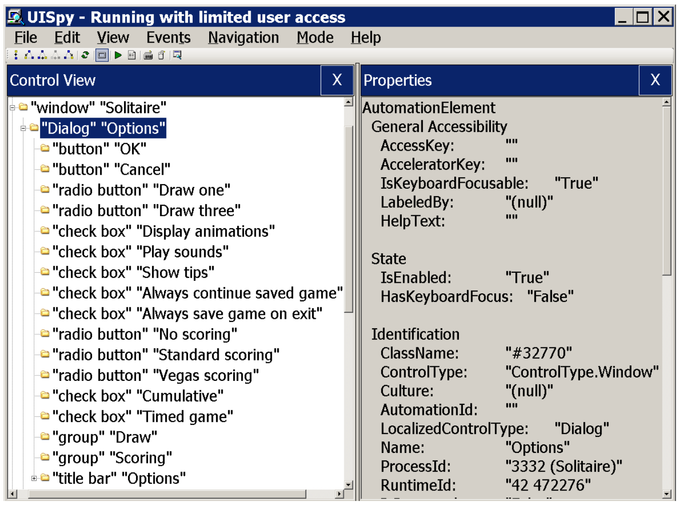
- ☑ Display animations
- ☑ Play sounds
- ☑ Show tips
- ☐ Always continue saved game
- ☐ Always save game on exit

Scoring
- ◯ No scoring
- ● Standard scoring
- ◯ Vegas scoring
- ☐ Cumulative

- ☑ Timed game

OK Cancel

Fig. 4. The options dialog of the Solitaire game.



UISpy - Running with limited user access
File Edit View Events Navigation Mode Help
Control View
Properties

```
"window" "Solitaire"
  "Dialog" "Options"
    "button" "OK"
    "button" "Cancel"
    "radio button" "Draw one"
    "radio button" "Draw three"
    "check box" "Display animations"
    "check box" "Play sounds"
    "check box" "Show tips"
    "check box" "Always continue saved game"
    "check box" "Always save game on exit"
    "radio button" "No scoring"
    "radio button" "Standard scoring"
    "radio button" "Vegas scoring"
    "check box" "Cumulative"
    "check box" "Timed game"
    "group" "Draw"
    "group" "Scoring"
    "title bar" "Options"
```

```
AutomationElement
  General Accessibility
    AccessKey:            ""
    AcceleratorKey:       ""
    IsKeyboardFocusable:  "True"
    LabeledBy:            "(null)"
    HelpText:             ""

  State
    IsEnabled:            "True"
    HasKeyboardFocus:     "False"

  Identification
    ClassName:            "#32770"
    ControlType:          "ControlType.Window"
    Culture:              "(null)"
    AutomationId:         ""
    LocalizedControlType: "Dialog"
    Name:                 "Options"
    ProcessId:            "3332 (Solitaire)"
    RuntimeId:            "42 472276"
```

Fig. 5. A Screenshot of "UI Spy" [10] showing the UIAutomation tree of the options dialog. (The font size was enlarged for the screenshot.)

Display handlers converting the bitmap into appropriate pin state definitions can be registered with the renderer. Of course, for the BrailleDis 9000 this is provided with the framework. In our implementation we have used a special bitmap format encoding pixels originating from Braille output and blinking pixels in addition to the pin up or down information. We also provide a graphics library to handle this special format and convert it to ordinary black and white bitmaps. The Braille encoding mainly was realized to support sighted developers with a readable view of the tactile output. But it can also be used to provide speech output for the tactile user interface. In the following chapter, we have used the developer's view to present some of our results. This view is synchronized with the tactile output supporting the communication between sighted and visually impaired users.

For the implementation of the interface components, we have utilized Windows.Forms. This provides everything we needed. By inheriting from Windows.Forms.Control [3] we can easily build up a widget hierarchy and also use pre-defined layout controls of the .NET Framework. Using the `DrawToBitmap` method, the tactile user interface can be rendered to a bitmap without presenting it on the screen. Furthermore, developers can test their widgets' rendering behaviour with an ordinary Windows.Forms project.

For the default rendering of common user interface elements, like buttons, menu items and labels, our framework provides implementations of the widgets mentioned in section II. Thereby, the widget modules used to mark the element type are realized by our interpretation of tactile icons, so-called tactons [13], for tactile graphics displays. Our tactons are fixed combinations of lifted and lowered pins that always have to be shown in full size and may not be scaled or cut. So the tactile impression always stays the same and serves to identify the widgets. For the different borders to present enabled and disabled state or special groupings an implementation of dash patterns is provided. Those patterns can be defined separately for each of the four borders (top, left, right and bottom). Furthermore, layout algorithms were implemented, to align widgets according to the original layout but at the same time generate a compact presentation that requires as less scrolling as possible. Of course, our framework is not bound to the default widgets. New widgets can be integrated easily.

B. Influencing the Rendering Result

The rendering result currently can be influenced in different ways. One way is to change the tactons used as markers for the different widget types. This can easily be done via replacing the tacton library used for rendering, which is a simple XML file. There the tactons are defined by patterns of '0' for lowered pins and '1' for lifted pins. A name is given to identify each tacton. Changing the tacton library influences the rendering of each user interface. It is meant to easily adapt the "Look&Feel" of the tactile user interface representations to a single users preferences or to new findings from user studies. It was kept as simple as possible so that it can also be changed by blind people themselves.

Of course, replacing tactons can influence the rendering only literally. The layout of the interface remains the same. Furthermore, it is also not possible to overcome filtering problems. For example, a clickable label will be reported as label by a UIAutomation filter and not as button. So for this label the press marker will not be used, although it should be. Changing the layout and implementing special rendering for single interface components is possible by creating own rendering methods. Such rendering methods can be registered with the renderer via the event handling mechanism of the .Net Framework. In those methods own widgets can be created and used. They only have to be derived from Windows.Forms.Control and implement the `Paint` method to provide an appropriate bitmap output for the tactile rendering. The event handling mechanism was used for the registration as it is the most flexible way and applies almost no restrictions. Furthermore, it is compatible with the Add-In pipeline that is used in the HyperBraille software and based on the Add-In-architecture of Microsoft .NET [2]. Self-defined rendering methods can also retrieve the widget representation of a certain object as it would be if this method was not registered. So, small changes to the existing rendering can be applied easily. That means, if for example someone just wants to change the representation of a label within a dialog box, it is not necessary, to implement the rendering for the whole dialog, but only for the label to change. The other dialog elements as well as the layout of the elements will be provided by the default rendering of our framework. The interface defined for the rendering methods is not restricted to a special input type. The rendering can be adapted to whatever content the application filters generate. Be it a UIAutomation tree or a string including the serialization of a graphical user interface.

Furthermore, the Braille output can be influenced easily, even without programming. The Braille font used for rendering is also defined in a simple XML file that assigns unicode characters to pin state patterns. So one can easily switch between different languages or define Braille signs for special characters in mathematical expressions. As Braille text is quite large related to the size of tactile displays, we also implemented the possibility to shorten text easily. Therefore, another XML file was defined containing pairs of text patterns to search for and appropriate replacements. So one can, for example, shorten "Replace with" by "Repl. with" or "User manual" by "Help". This saves much space and so helps to reduce the scrolling effort.

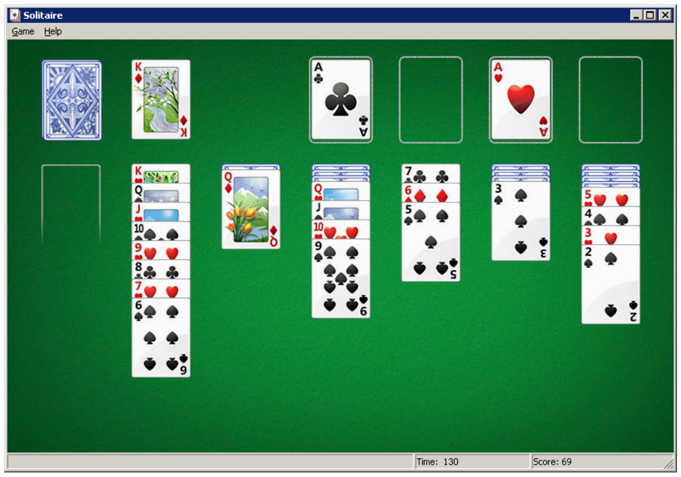


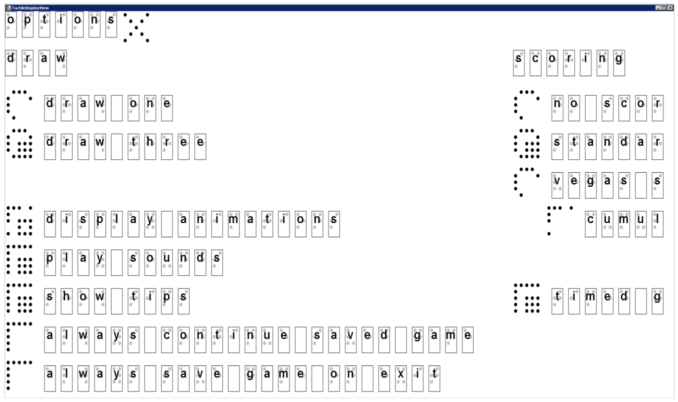
Fig. 6. The graphical user interface of the Windows Solitaire game.

C. Input Handling

Our framework not only handles different displays as output but also as input devices. Therefore, display handlers can be connected that provide the renderer with a scrolling and a click event. The scrolling is used to move the output of the tactile user interface on the display. Of course, not every user interface will fit on the display at once. The click event is used to control the rendered graphical desktop application via the tactile graphics display. Besides text input, click handling is sufficient for controlling the majority of graphical desktop applications. Every user interface element that was converted into a part of the tactile user interface can be clicked and the click will be forwarded to the original application. Therefore the tactile display handler has to provide the renderer with the coordinates of the clicked pin. The renderer then retrieves the widget of the tactile user interface that was rendered to this pin and requests the screen coordinates to click on for this widget. In our implementation these coordinates are set for the widget during its creation in the rendering method. So, the original user interface element does not have to be analyzed once more.

IV. Results

In the following, we use the Solitaire game (see fig. 6) to illustrate how the framework eases rendering a graphical desktop application to a tactile display. To analyze the elements of the user interface, we have used a generic UIAutomation [9] filter that is part of the HyperBraille software and converts UIAutomation elements into components of the HyperBraille off-screen-model [16].


options

draw

scoring

draw one

no scor

draw three

standar

vegas s

display animations

cumul

play sounds

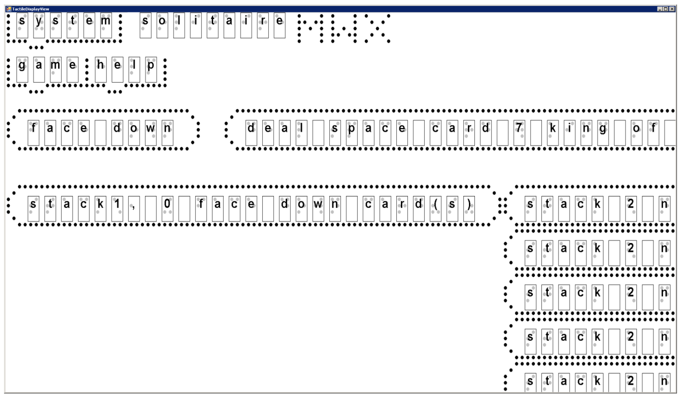
show tips

timed g

always continue saved game

always save game on exit

Fig. 7. The tactile representation of the options dialog of the Solitaire game. The close button, radio buttons and check boxes are rendered with tactons. The buttons ”OK” and ”Cancel” do not fit on the display but can be scrolled into view. This can be communicated to the user via optional tactile scrollbars.


system solitaire

game help

face down

deal space card 7 king of

stack1, 0 face down card(s)

stack 2 n

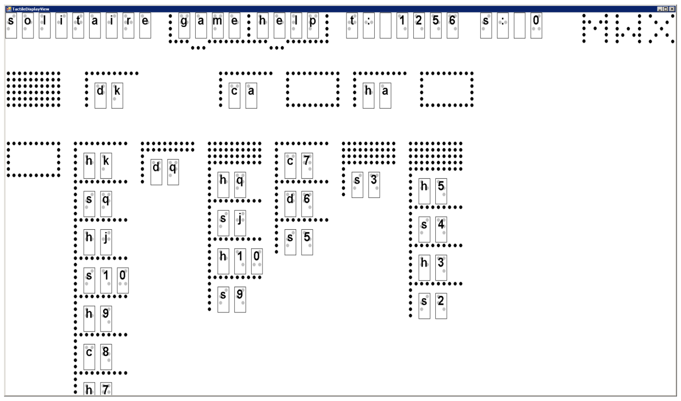
stack 2 n

stack 2 n

stack 2 n

stack 2 n

Fig. 8. The framework’s default tactile rendering of the Solitaire game. Note that the screenshot shows only a part of the whole user interface as only a few elements of the interface fit on the tactile display at once in the default rendering. To grasp the whole interface much scrolling is necessary.


solitaire game help t: 1256 s: 0

dk ca ha

hk sq hj s10 h9 c8 h7

dq

hq sj h10 s9

c7 d6 s5

s3

h5 s4 h3 s2

Fig. 9. The optimized tactile rendering of the Solitaire game which needed less than 400 lines of code. Almost the whole user interface fits in one display view. Scrolling is only necessary if the stacks of cards get very huge.

As can be seen in figure 7, showing the tactile output for the options dialog of the Solitaire game, the default rendering of our framework produces quite good results for ordinary dialogs with standard controls. For rendering check boxes and radio buttons, we have used instances of the `TactonWidget` that is provided with our framework. Unfortunately, the borders of the groups are not visible in the tactile rendering as the groups are not defined via hierarchical composition of the group and the items they visually contain. So the information about the grouping is not available to the UIAutomation filter. This can also be seen in figure 5. Although, the groups are also represented in the UIAutomation tree (e.g. in the line ”group” ”Draw”), the items that are visually positioned within the groups are not children of the group elements. Also for the layout of the tactile user interface only the position of the elements can be used.

The default rendering of the Solitaire game itself is shown in 8. It shows that depending on the tactile display’s size, much scrolling and significant cognitive effort is needed to grasp the whole playing field. The UIAutomation filter reports all the cards in the game as buttons and panes containing the description of the cards in their names. As the default rendering uses the buttons’ names as captions for button widgets, it does not really result in an interface that supports users of the display. Of course, it is possible to interact with the resulting interface and so with the original application also with this rendering. But because of the long button texts it is quite hard to keep the presented cards in mind. Furthermore, it may also be confusing, that the cards are sometimes represented as buttons and sometimes as panes, although they can be clicked in both states. So for the Solitaire game itself, special rendering methods have to be provided to turn it into a useful tactile representation.

With just one day of work and 400 lines of code for special rendering methods, we have managed to realize a really useful representation for our 120×60 pins sized tactile graphics display. Thereby most of the time was spent in investigating the UIAutomation output of the game to figure out, how the different buttons and panes are named in different situations. This was necessary to create an optimized layout. Therefore, we have not only changed the rendering of the cards themselves, but also adjusted spacing between different areas of the playing field. Furthermore, we have rearranged the title, the menu and the status bar to fit into one single line. This illustrates how useful it can be to be independent of the original user interface’s layout. As you can see in figure 9 utilizing the new rendering methods the whole user interface fits onto the display at once. Scrolling is only necessary if the card stacks get really large.

To demonstrate how easy it is to use the framework, we finish this section by providing some excerpts of the code that is used for the rendering of the Solitaire game. The following lines realize the nice rendering of the cards. They are part of a small C# class, that in it’s constructor registers the rendering methods with the renderer that is provided by the framework as singleton. This automatically calls the rendering methods, puts the returned widgets into the widget hierarchy and creates the bitmap that is sent to the tactile display.

```
// Define shortenings for the card types.
```

```
Dictionary<String, String> cardTypes
  = new Dictionary<String, String>() {
  {"Hearts", "h"}, {"Spades", "s"},
  {"Clubs", "c"}, {"Diamonds", "d"}};

// Shortenings for the values of the cards.
Dictionary<String, String> cardValues
  = new Dictionary<String, String>() {
  {"two of ", "2"}, {"three of ", "3"},
  ..., {"king of ", "k"}, {"ace of ", "a"}};

// Rendering methods are defined as event
// handlers. They can access the renderer
// via the sender object. The content to
// render is given in the GetWidgetEventArgs.
// The widget for the tactile user interface
// is defined via assigning it to e.Widget.
// If e.Widget is left undefined, other event
// handlers are called with the same
// rendering content.
void WidgetForCard(object sender,
                   GetWidgetEventArgs e)
{
  // this method handles only buttons
  if (e.ToRender is Button)
  {
    String text = String.Empty;
    // Get the name of the button.
    // This contains the card's type and value.
    String name = (e.ToRender as Button).Name;

    // Retrieve the shortenings to represent
    // the card's type and value.
    foreach (var kvp in cardTypes) {
      if (name.IndexOf(kvp.Key) > -1) {
        text = kvp.Value; break;
      }
    }
    foreach (var kvp in cardValues) {
      if (name.IndexOf(kvp.Key) > -1) {
        text = text + kvp.Value; break;
      }
    }

    // If the button really represented a card,
    // create a TextWidget with the shortenings
    // as text, a surrounding border and the
    // screen coordinates of the card as click
    // point as widget of the tactile user
    // interface.
    if (text.Length > 0)
      e.Widget = new TextWidget(text) {
          TopBorder = new SolidBorder(),
          LeftBorder = new SolidBorder(),
          Size = new Size(10, 6),
          Tag = GetClickPoint(e.ToRender)
      };
  }
}
```

Here we have used the class TextWidget that is provided by the framework for rendering text into Braille surrounded by a tactile border line. You can also see that the programmer does not have to care about the overall layout. Although, it can be influenced. But if the default layout is satisfying, it also does not have to be reproduced when changing some small parts of the rendering.

V. Evaluation

The rendering part of our framework is used to create the graphical tactile rendering of the HyperBraille software - the HyperReader. Thereby, it has been tested by at least ten developers everyday since one and a half year. A number of Add-Ins have been implemented by the project partners that register rendering methods to apply special rendering for applications like Visio (special rendering for UML shapes) and Excel (special rendering for formulas).

For the development of the Add-In methods the framework was often used independently of the HyperReader. So the framework proved to be useful for a kind of rapid prototyping for tactile representations of desktop applications.

Naturally, all adaptations to the default rendering were presented to our blind project partners and also other blind subjects interactively on the BrailleDis 9000 and were improved on basis of their comments. User evaluations showed that representations should be as compact as possible. Representations should always try to avoid scrolling (especially horizontal scrolling) as far as possible just like shown in the Solitaire example. Also the configurable text shortenings have been accepted very well as they support compact layout. For known GUIs users often only need to read a couple of the first characters of an element's label to know which element is shown. Blind users performed configurations for text shortenings and type markers (tactons) by themselves.

VI. Conclusion and Future Work

We have presented ongoing work on a framework for running graphical desktop applications on tactile graphics displays. The example of Solitaire illustrated how easy it is to use the framework and adapt the default rendering for special components. Only a few lines of code are necessary to achieve really good results. We have shown that also for tactile user interfaces the use of widgets and layout routines is valuable as thereby applying small changes to a default rendering is easy.

As the rendering interface is open to arbitrary objects, it is possible to handle any application. The developer only has to provide an appropriate filter to analyse the user interface and rendering methods to handle the filter's output. Of course, analysing the graphical user interface is easier the more accessible it is.

We believe that our framework pushes the work on tactile graphics displays one step forward from the software engineering view. Tactile graphics displays get more useful the more applications can be controlled with them. As the framework implements a set of default widgets and layout algorithms a number of ordinary desktop applications can be translated to tactile user interfaces easily. Furthermore, a common Look&Feel is promoted for tactile user interfaces, which makes it easier for users to cope with new applications. Of course, our framework cannot create an optimal tactile rendering for each and every application automatically. But as it will always produce a representation for each interface element, it reveals the possibility that the users themselves can adjust the rendering by creating own rendering methods. So blind people can play a very active role in the design process of tactile user interfaces, just as they can currently improve the output of ordinary screen readers.

In the future, we want to provide more predefined widgets that can be configured and mapped to GUI elements without any line of code but just by XML configuration files in an even more flexible way than it is possible now. Of course, our framework is integrated in the graphical screen reader that is developed in the HyperBraille project and so will be available with this in the future.

Acknowledgments

We thank our colleagues of the HyperBraille project for their support in the design of the rendering framework and the design of the tactile output itself. The HyperBraille project is funded by the German Ministry of Economy and Technology (BMWi) under the grant number 01MT07004. Only the authors of this paper are responsible for its content. The Consortium consists of Metec AG, F.H. Papenmeier, Human-Computer Interaction Research Group at the Technical University of Dresden, Visualization and Interactive Systems Institute, IZFM institute and IMS-Chips at the University of Stuttgart, the Research Group for Information Processing and Communication at the University of Potsdam, as well as Blista Marburg and T.O.P. GmbH & Co. KG Freiburg.

References

[1] HyperBraille: The graphics-enabled display for blind computer users (supported by BMWi and DLR). http://www.hyperbraille.de/.

[2] MSDN Library: Add-ins and Extensibility. http://msdn.microsoft.com/en-us/library/bb384241.aspx.

[3] .NET Framework Class Library: Control Class. http://msdn.microsoft.com/en-us/library/system.windows.forms.control.aspx.

[4] RNIB National Centre for Tactile Diagrams. http://www.nctd.org.uk/.

[5] TactileGraphics.org. http://www.tactilegraphics.org/.

[6] The Blind Readers' Page - Tactile Maps and Graphics. http://www.blindreaders.info/mapgraph.html.

[7] The Reginald Phillips Research Programme - investigating tactile graphics in the education of blind children. http://www.lifesci.sussex.ac.uk/reginald-phillips/.

[8] Touch Graphics Research. http://www.touchgraphics.com/research.htm.

[9] Windows Developer Center: Accessibility Overview. http://msdn.microsoft.com/en-us/windows/bb735024.aspx.

[10] Windows Presentation Foundation Tools: UI Spy (UISpy.exe).

[11] K. Albert. Konzeption und Entwicklung eines taktilen User Interfaces (Design and Development of a Tactile User Interface). Großer Beleg, Institute of Applied Computer Science, TU Dresden, 2008.

[12] F. K. Aldrich and L. Sheppard. Tactile graphics in school education: perspectives from pupils. *British Journal of Visual Impairment*, 19(2):69–73, 2001.

[13] S. A. Brewster and L. M. Brown. Tactons: Structured Tactile Messages for Non-Visual Information Display. In A. Cockburn, editor, *AUIC*, volume 28 of *CRPIT*, pages 15–23. Australian Computer Society, 2004.

[14] K. Itou, B. Kato, M. Taniguchi, T. Otogawa, K. Itoh, K. Kiyota, N. Ezaki, and K. Uchimura. Learning Support System Based on Note-Taking Method for People with Acquired Visual Disabilities. In Miesenberger et al. [19], pages 813–820.

[15] M. Kobayashi and T. Watanabe. Multimedia Communication System for the Blind. In C. Ferreira, editor, *Intelligent Paradigms for Assistive and Preventive Healthcare*, volume 19 of *Studies in Computational Intelligence*, pages 165–181. Springer, 2006.

[16] M. Kraus, T. Völkel, and G. Weber. An Off-Screen Model for Tactile Graphical User Interfaces. In Miesenberger et al. [19], pages 865–872.

[17] M. Miao, W. Köhlmann, M. Schiewe, and G. Weber. Tactile paper prototyping with blind subjects. In *HAID '09: Proceedings of the 4th International Conference on Haptic and Audio Interaction Design*, pages 81–90, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] K. Miesenberger, J. Klaus, and W. L. Zagler, editors. *Computers Helping People with Special Needs, 8th International Conference, ICCHP 2002, Linz, Austria, July 15-20, Proceedings*, LNCS. Springer.

[19] K. Miesenberger, J. Klaus, W. L. Zagler, and A. I. Karshmer, editors. *Computers Helping People with Special Needs, 11th International Conference, ICCHP 2008, Linz, Austria, July 9-11, 2008. Proceedings*, LNCS. Springer.

[20] D. Prescher. Ein taktiles Fenstersystem mit Multitouch-Bedienung ((A tactile window system for multitouch operation). Diploma thesis, Institute of Applied Computer Science, TU Dresden, 2009.

[21] M. Rotard, K. Otte, and T. Ertl. Exploring Scalable Vector Graphics for Visually Impaired Users. In J. Klaus, K. Miesenberger, W. L. Zagler, and D. Burger, editors, *ICCHP*, LNCS, pages 725–730. Springer, 2004.

[22] M. Rotard, C. Taras, and T. Ertl. Tactile web browsing for blind people. *Multimedia Tools Appl.*, 37(1):53–69, 2008.

[23] M. Schiewe, W. Köhlmann, O. Nadig, and G. Weber. What You Feel Is What You Get: Mapping GUIs on Planar Tactile Displays. In C. Stephanidis, editor, *HCI (6)*, volume 5615 of *LNCS*, pages 564–573, 2009.

[24] L. Sheppard and F. K. Aldrich. Tactile graphics in school education: perspectives from teachers. *British Journal of Visual Impairment*, 19(3):93–97, 2001.

[25] F. Vidal-Verdú and M. Hafez. Graphical Tactile Displays for Visually-Impaired People. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 15(1):119–130, March 2007 2007.

[26] T. Völkel, G. Weber, and U. Baumann. Tactile Graphics Revised: The Novel BrailleDis 9000 Pin-Matrix Device with Multitouch Input. In Miesenberger et al. [19], pages 835–842.

[27] T. Watanabe, M. Kobayashi, S. Ono, and K. Yokoyama. Practical use of interactive tactile graphic display system at a school for the blind. In A. Mendez-Vilas, A. S. Marti, J. M. Gonzalez, and J. M. Gonzalez, editors, *Technological Science Education, Collaborative Learning, Knowledge Management*, volume 2 of *Current Developments in Technology-Assisted Education*, pages 1111–1115. FORMATEX, 2006.