AwToolkit: Attention-Aware User Interface Widgets

Juan E. Garrido, Victor M. R. Penichet,
Maria D. Lozano
Computer Science Research Institute
University of Castilla-La Mancha
Albacete, Spain

Aaron Quigley, Per Ola Kristensson
School of Computer Science
University of St Andrews
United Kingdom

ABSTRACT
Increasing screen real-estate allows for the development of applications where a single user can manage a large amount of data and related tasks through a distributed user interface. However, such users can easily become overloaded and become unaware of display changes as they alternate their attention towards different displays. We propose AwToolkit, a novel widget set for developers that supports users in maintaing awareness in multi-display systems. The AwToolkit widgets automatically determine which display a user is looking at and provide users with notifications with different levels of subtlety to make the user aware of any unattended display changes. The toolkit uses four notification levels (unnoticeable, subtle, intrusive and disruptive), ranging from an almost imperceptible visual change to a clear and visually salient change. We describe AwToolkit's six widgets, which have been designed for C# developers, and the design of a user study with an application oriented towards healthcare environments. The evaluation results reveal a marked increase in user awareness in comparison to the same application implemented without AwToolkit.

Categories and Subject Descriptors
H.5.2. [**Information interfaces and presentation**]: User Interfaces—*Windowing systems*

General Terms
Design, Human Factors.

Keywords
Attention, awareness, notifications, distributed user interfaces.

1. INTRODUCTION
Complex environments often require systems and applications that involve controlling large amounts of data. Such applications can involve interfaces that are spread across multiple displays that a user must deal with. As a result, the number of tasks and events a user needs to manage or supervise can become overwhelming or require a user to divide their attention (e.g. [1, 2, 3, 4, 5, 6, 7, 8]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
AVI'14, May 27–29, 2014, Como, Italy
Copyright 2014 ACM 978-1-4503-2775-6/14/05$15.00.
http://dx.doi.org/10.1145/2598153.2598160

Healthcare centers [4] represent an example of such a complex environment. Generally, operators have to ensure that all the tasks are performed on time, and they have to assign new tasks and manage emergencies. To this end, an operator typically controls a system deployed on multiple displays that are continuously revealing changing information. However, new display content might be missed by the operator because it can suddenly appear within a large amount of data, while the operator is not looking at the screen. The importance of unnoticed changes depends on the nature of the task but on the whole, the loss of information can be considered a disadvantage. For example, in healthcare environments, lost information may result in the incorrect treatment of patients, which may then affect their health.

In this paper we present AwToolkit, as an approach to the delivery of updates and notifications from unattended displays. AwToolkit consists of a set of user interface widgets (AwWidgets) that assist users in maintaining awareness of display changes using different levels of interruptions of their current task. These AwWidgets are a class of gaze-aware user interface components inspired by the recent work on subtle gaze-dependent techniques for visualising display changes in multi-display environments [3]. The main objective of AwWidgets is to offer developers the capability of programmatically detecting changes in parts of the application, when users are not looking at a specific display, and then notifying users of these changes using a notification system that supports different degrees of subtlety. We present six user interface widgets (AwPanel, AwButton, AwLabel, AwTextBox, AwComboBox and AwPlayer) as examples of specific implementations of the AwWidget toolkit concept.

2. RELATED WORK
Gaze-aware user interface components are a largely an unexplored research area. Existing work focuses on providing users with appropriate information on events to help them with their tasks. We describe some of the most relevant studies, which are relevant to the work and concepts presented here.

The MAUI toolkit [6] is a pioneering project that provides a set of widgets oriented towards group work, based on ex-

isting standard user interface widgets. This toolkit can collect, distribute and visualize group awareness information in widgets. In addition, MAUI provides a variety of both standard and groupware interface components in Java. The use of the toolkit results in improvements in collaborative environments [6]. However, the toolkit does not focus greatly on building collaborative group-aware interfaces. Specifically, MAUI offers users some types of awareness but the main one only reveal people's activities as they work with the application as is not related to their gaze.

Mnemonic rendering [2] involves buffering changes in pixels and restoring them to the screen. It is primarily based on the principle of visible pixels, which may sometimes be hidden. Consequently, Mnemonic rendering ensures the storage of changes in hidden pixels to avoid losing visual information at a window level. These pixels will then be restored by the history buffer, thereby revealing the changes that happened while they were not visible to the user. Based on user's gaze, TeleEye [8] provides information regarding a sense of presence and workspace awareness [5]. The objective is to reveal the location of the user's attention using eye tracking in a collaborative setting. One of the main advantages provided in TeleEye applications is the possibility of improving coordination and communication of actions between users.

DiffDisplays [3] is a system that is able to track the display the user is looking at by using a series of web cameras in a multi-display environment. It helps users to quickly gauge what has changed since they last looked at a display. DiffDisplays uses four different high-level visualization techniques to highlight changes. These techniques are executed when you look at a display, after a period of initially not looking at it. Using the entire screen or windows, the visualizations aim to provide the user with an overview (or an awareness) of changes that happened while their attention was focused elsewhere.

The toolkit we propose in this paper is based on the published DiffDisplays codebase [3]. The main difference with respect to related work is that AwToolkit is a toolkit for developers based on tracking the user's gaze to produce notifications with different levels of subtlety in order to inform users about changes at the widget level. In contrast, MAUI provides a toolkit for group work but it does not take into account the user's gaze when generating information or the changes in the interface. The primary difference to mnemonic rendering is that it does not work at the widget level but at the window level. Finally, while employing the user's gaze when generating information, Teleye ignores changes in the interface which might be useful to inform the user of.

3. AWTOOLKIT

Gaze-aware user interface components (widgets) and subtle change within these widgets is a largely unexplored area. We categorize widgets to help developers generate change notifications when users return their gaze to a display that has changed. In this way, the widgets can be classified into three categories that take into account a minimum level of disruption. The first category involves widgets that act on the entire screen. The DiffDisplays [3] system, described previously, represents an example of this. The second category refers to widgets which provide information about changes but focus on a specific part of the screen. AwWidgets is an example of this. It consists of a set of user interface widgets which we will present in detail later in this paper. Finally, the third category refers to mirroring techniques that notify the user of changes on the screen the user is looking at, which correspond to updates that have appeared on an unattended display.

AwToolkit is a toolkit implemented in the C# programming language that aims to provide developers with gaze-sensitive user interface controls. AwToolkit adds six new user interface controls to the programming environment based on the functionality of some of the already existing controls: *panel*, *button*, *label*, *textbox*, *comboBox* and *media player*. This new functionality consists of the capability to detect whether the user is looking at the screen where the widget is located, and the ability to react to such an event in an appropriate fashion.

The design of each user interface widget is based, in general terms, on the approach a typical Visual Studio C# developer might use the controls. Each widget is customisable in appearance and behavior. The appearance can be modified through the associated properties, while the behavior can be controlled and managed via the events related to each widget.

A key objective in designing AwWidgets has been to preserve the look and feel of Visual Studio. To this end, the toolkit is encapsulated into a library which after it has been imported will add the widgets as new tools in the Visual Studio toolbox.

3.1 General Features

The key feature of AwToolkit is the capability of the widgets to detect the user's gaze and consequently show changes when the user's gaze returns to a previously unattended widget screen. The detection procedure is based on the Diff Displays system [3]. Each widget is connected to a server provided by the Diff Displays [3] system that is continually sending information that indicates whether the user is looking at a specific display or not. The information supplied by the server is configured so as to know, second by second, if the user is looking at a particular screen. Currently, in the case of having more than one screen, the server must be run for each one. When configured, the server constantly sends information related to the user's gaze so that it can be read by the application that requires it.

Each widget can generate two gaze-aware states: *advice* and *normal*. The first state refers to a state in which the user is not looking at the screen that contains the widget and one or several display changes have taken place on the screen. If the widget is not in the advice state it will be in the normal state, indicating that the user is looking at the screen containing the widget and/or there are no display changes.

The way display changes are shown to the user will depend on the environment created by the developer for each application. Each environment will offer certain conditions and requirements which involve an adjustment of the level of subtlety of the information the user receives regarding changes.

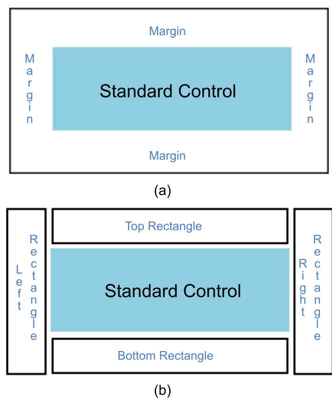

Figure 1: AwWidgets layout and structure.

Consequently, a set of modes has been defined to specify the level of subtlety of information on a change:

- *Disruptive*: the user is alerted and the current activity is disrupted.
- *Intrusive*: the user is alerted and we can expect their gaze to be directed to the location of the display change. However, the user can freely continue their current activity.
- *Subtle*: the user is alerted more lightly and their attention is attracted to the display change. However, their gaze is only directed to the location of the display change.
- *Unnoticeable*: the user is alerted so subtly that if they are not paying attention they may not notice the display change signal.

In order to customize the level of subtlety, the developer is provided with a set of elements which modify the visual appearance and which can be maintained, individually or jointly, for a determined period of time. Each widget contains specific visual elements which indicate changes in the widget itself. These elements are displayed when one or more changes have taken place and the user is not looking at the screen. When the user looks at the widget screen, the visual change elements disappear after a certain time. The time they take to disappear is defined by the property specified for this purpose. In general terms, the way the developer defines what visual marks appear and for how long, will determine the level of subtlety of the information the user receives regarding the changes.

The customization of the level of subtlety is a result of the properties created for the purpose of each widget and which facilitate the desired level of subtlety: *properties configuring which display change notifications that will be used with an AwWidget* (if they are not part of the normal appearance as is the case in AwComboBox), *properties to indicate the appearance of the graphic elements of the change notification,*

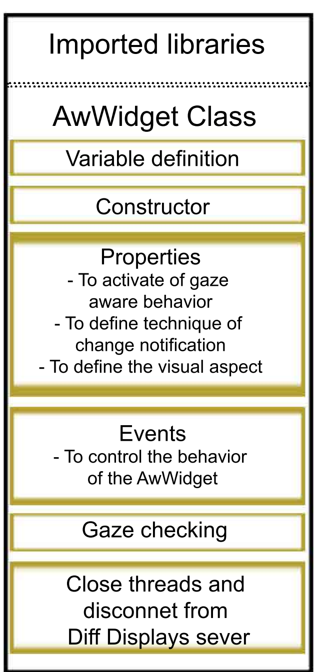

Figure 2: Structure of AwWidgets.

and *AwTime* (to indicate the length of time change notification elements are to be displayed). However, in general terms the authors have established a property (*AwDefaultTechnique*) which automates level selection. This property makes a default selection of visual change elements and durations for each widget in accordance with each level of subtlety. The defaults are based on the authors' experience and the comments and opinions of the users participating in our evaluation, described later in this paper.

The final feature of AwToolkit is the localization of changes in the content and functionality of the widgets. Three methods have been used for detecting changes, based on the type of widget. The first method is based on successive screenshots of the widget and, by definition, its visual content, which represents the evolution over time. The second method uses the widget text, which is a default visual element and is represented by a property. The third method is based on the modification of any specific state-changing functionality of the widget, such as for example a button changing its disabled state to an enabled state when the user is not looking.

3.2 Implementation Details

In terms of C#, each AwToolkit widget shares a specific structure (see Figure 1 for one example aspect). Figure 2 shows the overall structure for a widget. Due to the shared structure, reusing some parts for the development of a new widget is an easy task, which helps such a development; therefore, extending AwToolkit with new components does not involve many new lines of code.

First, the widget structure imports libraries: (1) to use threads in the implementation of the parallelization of gaze

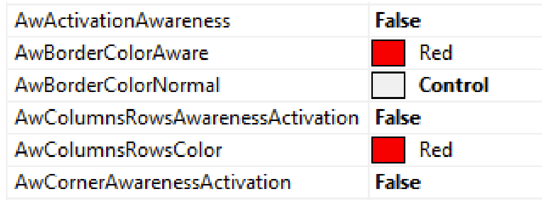

Figure 3: Example properties of an AwWidget to define the method of notifying display changes.

checking, running the common functions of the control and modifying the interface look based on the user's gaze; (2) to use sockets to connect with the Diff Displays [3] server which provides information about whether the user is looking at the screen or is not; (3) to be able to modify the visual control look; and (4) to be able to close external processes, closing the Diff Displays [3] server once the application has exited and no more resources are required.

Next, we have the code regarding the class that constitutes each AwWidget. Such a class extends another one called *UserControl*, which provides the basic functionality of a new control. In this way, the control may be used as another graphical item in the toolbar in Visual Studio. The AwWidget properties (Figure 3) allow the user: to select the behavior of the widget based on the user's gaze; to define what technique to use to show the user every change in the system which took place when and where they were not looking at the widget; and to define the visual look of the widget. The visual look will be different if (a) the user was looking and there were no changes; or if (b) the user was not looking and there were changes. In the second case, the user will be warned about the changes by means of notifications with different levels of subtlety defined by the developer.

After variables, constructor and properties, the structure of the class provides events related to the widget, which are defined as functions that are called when an event occurs. It is important to highlight the *OnPaint* event. It is called when then widget is drawn on the screen, such as during each screen refresh of the application. Since such event runs continually and involves re-drawing the widget, it is a suitable place to check the user's gaze as well as to look for changes to modify the "gaze-aware" look of the widget. To do so, the value of a variable with information of the user's gaze is checked. Such information is continually updated through a process running in parallel to the interaction of the user with the widget. Every AwWidget also offers a property called *ExistingChanges* which can be modified directly by the developer. Taking into account all this information, the widget properties are analyzed in the OnPaint method. If the user is not looking at the screen and any change in the control takes place (e.g. by disabling or changing functionality), visual components are added according to the selected technique. We will discuss this further in Section 3.3.

Items affected by users' gaze are found based on the information that the Diff Displays [3] server is continually transmitting. The server is configured to provide information every second, which is the same period that is used to check an AwWidget. The variable *gaze* provides information about

```
//It obtains the last package of diffDisplays
 answer = GiveMeLastDiffDisplaysMessage();
//We analyze the answer
if (answer.Contains("\"gaze\": 1") || answer.Contains
("\"gaze\": 2") ||  answer.Contains("\"gaze\":3 "))
{
     //Searching the "gaze" word
     for (int i = 0; i < answer.Length; i++)
     {
            if (respuesta[i] == 'g' && respuesta[i + 1]
           == 'a' && respuesta[i + 2] == 'z' &&
           respuesta[i + 3] == 'e')
            {
                    value_gaze = answer[i + 7].ToString();
                     if (int.Parse(value_gaze) == 1){
                     //The user does not look at the screen
                    lookAt = false;
                     }

                   if ((int.Parse(value_gaze) == 2)
                   || (int.Parse(value_gaze) == 3)){
                    //The user looks at the screen
                     lookAt = true;
                   }

             break;
            }
     }
}
```

Figure 4: Code to update the variable that indicates if the user looks at the screen.

whether the user is looking at a widget or not. According to its value, the variable *value_gaze* in AwWidget is updated, which in turn will influence the behavior and graphical look of the widget (see example code in Figure 4). All this activity connected to the gaze checking is implemented on a thread running in parallel to the functionality of the widget itself.

Finally, the last section of the structure of AwWidget concerns closing of threads and the disconnection of the Diff Displays [3] server to avoid unnecessary resource utilization.

The structure of AwWidgets makes it easy for developers to include them in their applications. Since the toolkit integrates with Visual Studio, once the toolkit has been imported, the only thing a developer needs to do is to drag the widgets from the toolbar and to drop them onto the visual interface on their application. The developer can change the properties of the widget and the events like any other standard control in Visual Studio. No extra code is required to make use of AwWidget when developing applications.

3.3 Widgets

AwToolkit currently offers developers six AwWidgets based on the features previously described: AwPanel, AwButton, AwLabel, AwTextBox, AwComboBox and AwPlayer. They are all based on a C# standard control but with new added characteristics. The modifications incorporate the capability of automatically detecting changes in control (content or functionality) or based on the user's gaze. If the user is not looking at the widget screen, changes are searched for, and if found, notifications are automatically presented to the user. The following sections describe the components of the toolkit according to the sharing mode used to notify the user of a change.

3.3.1 AwPanel

The first widget, AwPanel, can be considered an extension of an existing standard control: *the panel*. The main pur-

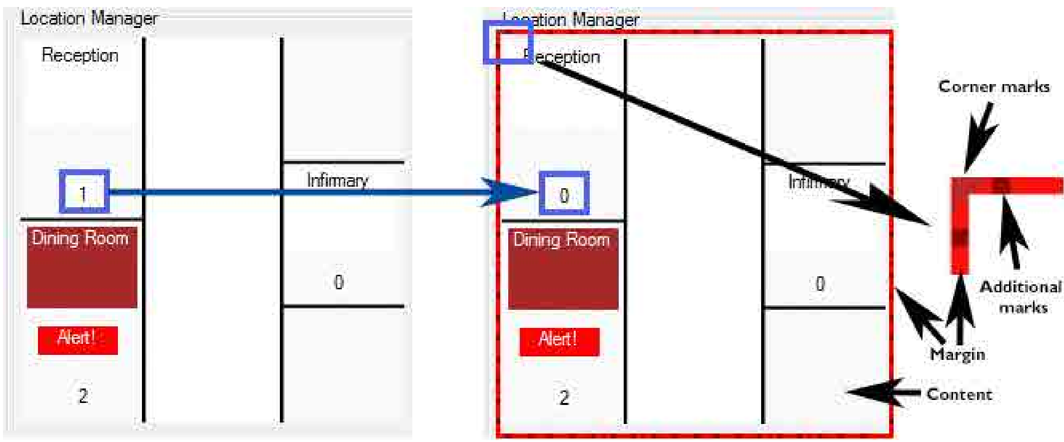

Figure 5: Example an AwPanel working in the Ubi4health system [4].

pose of this standard control is to help developers organize the content in an application. To this end, a panel of an application generates a specific area in which some elements are included. The panel appears as a polygon inside the application, the perimeter of which is marked by a continuous line.

The AwPanel widget detects changes by comparing screenshots over time. The widget captures a new screenshot regularly and the new screenshot is compared against the previous one. The comparison is performed on the pixel level.

The widget is composed of two main parts (see Figure 1a): *content* and *margin*. The first part is where the developer has to insert the section of the application, in which the user may wish to be aware of changes. The appearance of the margin area changes according to the widget behavior.

The developer has to make modifications on the margin of the panel to notify changes. A predetermined color in the margin represents the normal widget state of the widget. Then, if the user is not looking at the widget and changes are detected in the content area, the margin will be modified, changing from the normal state to the advice state (*ColorAware* property). Figure 5 shows an example of how AwPanel changes from one state to another. For this purpose, AwPanel offers three change notification techniques. The first two techniques are independent of the number of changes.

The first technique consists of modifying the widget margin by three individual or combined visual changes: *changing the color of the complete margin, inserting vertical and horizontal lines* and *inserting marks in the corners of the margin*. Properties determine the notification actions in the margin: *NormalMarginColor* (margin color in normal state), *AwareMarginColor* (margin color in advice state), *AwareMarksMargin* (marks added to the margin to notify changes). Obviously, if the visual elements are combined, each added element must be a different color so as to ensure a minimum differentiation. However, depending on the level of subtlety required, the difference between the color of the marks and the margin color can range from almost invisible to highly disruptive.

The second technique is an extension of the previous one. Here the panel is conceived as an area of the screen formed by four equally-sized squares. Each area displays a quadrant in a corner of the panel with different visual information. In this way, depending on which quadrant changes are produced in, a mark will appear in the relevant corner. This technique does not not only indicate that a change has taken place, it also gives some indication of where the change took place.

The third technique is called "*Book mode*" and it is dependent on a number of changes. If there is just one change the behavior is the same as the previous two techniques. Otherwise, the widget generates a *book* with the screenshot of each change. Therefore, when the user goes back to the screen, the widget shows the *book* as an overlay element of the panel. The *book* is a series of screenshots of the content and the user is able to navigate through them. The developer can rapidly generate a cursor control for the book thanks to the methods offered by the widget. Figure 6 shows an example of the Book mode.

3.3.2 AwButton, AwLabel and AwTextBox

This section describes three widgets that despite having different functionalities share the same approach to alerting the user of a change produced while they are not looking at the screen. The method in question is, in fact, the one described as the first technique in AwPanel (section 3.3.1), which modifies the margin color and which is also able to insert horizontal and vertical lines and marks in the corners. For this purpose, it is necessary to maintain the same structure as AwPanel in the distribution of the graphic elements into which AwButton, AwLabel and AwTextBox are divided. This structure is shown in Figure 1a.

AwButton is a refinement of the standard *button* control. The detection of changes is based on identifying the activation or deactivation of the functionality associated with pressing AwButton.

AwLabel extends the *label* control, which is a text description in a user interface. Since the main element of AwLabel is the text it displays to the user, the detection of changes

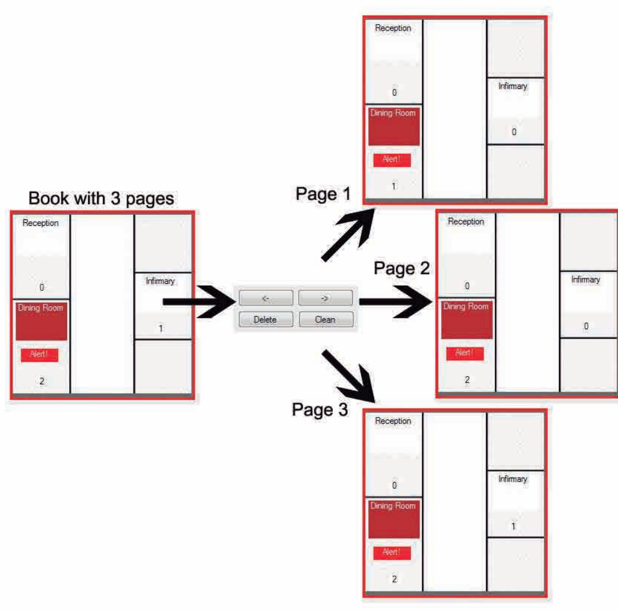

Figure 6: Visual description of AwPanel Book Mode.

is based on checking if the text has changed while the user is not looking at the widget screen.

Finally, AwTextBox defines a new version of the *TextBox* control, which is a widget with editable text. Similarly to AwLabel, the detection of changes focuses on identifying modifications in the text displayed by AwTextBox.

3.3.3 *AwComboBox*

AwComboBox is based on the standard ComboBox control, which shows a drop-down list of selectable items to the user. Taking into account the features of ComboBox, the detection of changes is based on: (1) the search for the elimination or insertion of elements in the drop-down list and (2) a change of the elements selected from the list.

The design of AwComboBox is different from that of the previous widgets. It has a drop-down design when the drop-down arrow is selected (see Figure 1b). It consists of the drop-down menu (standard ComboBox control) and four rectangles which border the widget. The aim of the four rectangles is to allow the developer to indicate changes based on their location. The initial idea of the use of rectangles is the following: (1) the upper and side rectangles indicate changes in the element selected from the drop-down menu (see Figure 7); (2) the lower rectangle displays changes in the list itself among elements which are not visible unless the list is dropped down; and (3) the four rectangles together offer changes in the non-visible list and in the selected element.

The widget completes its advice mode for changes produced when the user is not looking by the modification of the visual appearance of the rectangles. To this end, AwComboBox uses two individual or combined techniques: (1) changing the background color; and/or (2) adding horizontal lines inside the side rectangles and/or vertical lines in the upper and lower rectangles.

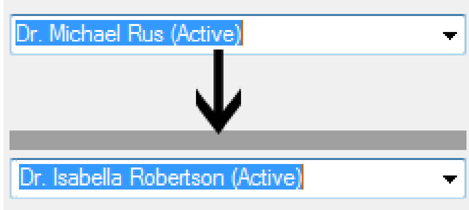

Figure 7: The way an AwComboBox shows changes.

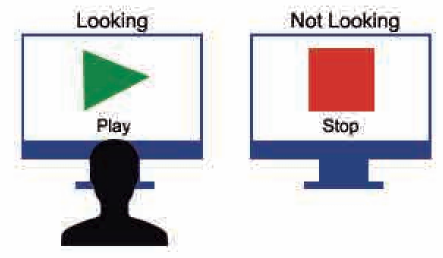

Figure 8: Behavior of AwPlayer.

3.3.4 *AwPlayer*

The last AwWidget is AwPlayer. If AwPlayer detects that the user has stopped looking at the screen, it stops playing back multimedia content. If it detects the user is looking at the screen again, it resumes playing. Figure 8 shows how reproduction progresses depending on the user's gaze. AwPlayer is designed for situations in which a user has the need to pay attention to all multimedia content.

A developer configures AwPlayer via the *PlayerSource* property. This property is the multimedia player on which AwPlayer acts. The developer inserts a multimedia player which will be the reproduction source on which AwPlayer will act depending on the user's gaze.

AwPlayer does not contain graphic elements since it already uses a multimedia player.

4. EVALUATION

AwToolkit has been evaluated from two perspectives. The first evaluation was with an application using AwWidgets used by end-users. The second evaluation elicited information from developers in order to better understand their fulfilled and unfulfilled needs and wants when developing with AwToolkit.

4.1 User's Point of View

The aim of this evaluation was to test how the developed widgets allowed users to be aware of screen changes in a specific environment. The evaluation was performed with 12 participants who worked with two versions of Ubi4health [4], a healthcare application with a three-screen configuration, which manages essential tasks in residential care homes. The first version incorporated AwPanel (in *book mode*), AwButton, AwLabel, AwTextBox and AwComboBox, while the

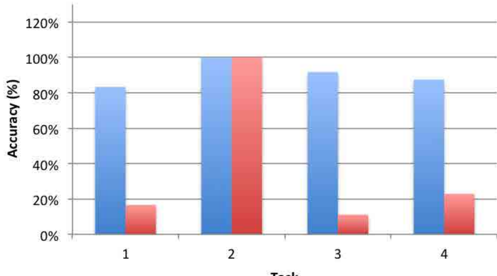

Figure 9: Average accuracy obtained during the evaluation with Ubi4health with AwWidgets (blue) and without (red).

second did not. The experiment was performed with three computers with screens, specifically two dual-core computers with 20-inch displays and one i-7 MacBook Pro connected to a 22.5 inch screen. Using a standard screen configuration, each computer executed one part of Ubi4health. The part that manages the tasks to be performed was on the left, the part that controls alerts was in the center, and the part that manages the staff was on the right. This is a typical use case and display configuration for this task.

Five participants were female and the other seven were male. Participants had an average age of 36; the oldest participant was 48 and the youngest was 21. All of them were familiar with computing systems.

The evaluation was based on the performance of four tasks that each user had to complete using the two versions of Ubi4health described. The tasks asked the participants to follow specific instructions and activities through two of the screens without viewing the third one. The users were invited to consider themselves as being responsible for the completion of pending tasks, staff organization and the immediate attention of emergencies in a real residential care home. Each user was introduced to the application through an initial tutorial in order to ensure they could understand the information given by Ubi4health and understand how to find possible changes in the tasks. The evaluation used four tasks:

- *Task* 1: the user has started their working shift and wants to take 15 seconds (ignoring *screen* 3) to check if all the tasks are progressing correctly (*screen* 1). At the same time, the user is constantly attending to possible resident alerts (*screen* 2). The user then decides to look at *screen* 3 to check where the staff are and reorganize them, as there are some tasks that have not been performed on time. Finally, the user has to indicate on *screen* 3 if there were changes, and if so, how many.
- *Task* 2: the user continues (for 15 seconds) working with *screen* 3 to organize the staff without looking at *screen* 2. Then, the user remembers that an emergency may appear and they look at *screen* 2. In that moment, the user has to indicate if there were changes on *screen* 2, and if so, how many.
- *Task* 3: the user looks again at *screen* 3 for 25 seconds to finish the staff reorganization but without forgetting *screen* 2 in order to control emergencies. Afterwards, the user should check the tasks not performed in time again, in case these have changed. Therefore, the user looks at *screen* 1 and has to indicate again if there were changes, and if so, how many.
- *Task* 4: the user analyzes the tasks (performed and not performed in time) to decide who are the most appropriate staff to complete the pending ones. After 15 seconds, the user looks at *screen* 3 to send messages to the correct employees. At this moment, the user has to indicate on *screen* 3 if there were changes, and if so, how many.

The evaluation was carried out as a within-subjects design with toolkit as the independent variable (using or not using AwWidgets) and accuracy as the dependent variable. The outcomes of the evaluation are shown in Figure 9, and indicate a clear improvement when using AwWidgets. AwWidgets resulted in significantly higher accuracy for Tasks 1, 3 and 4 ($p < 0.0001$; RM-ANOVA, Bonferroni corrected). These improvements are based on the accuracy of the participants' answers and the correctness of the participants' indications regarding the number of changes they have found. When using Ubi4health without AwWidgets the participants had a very low accuracy. There is only one task in which the answers about changes were correct, the one related to emergencies in which Ubi4health offers a clear visual change on the screen. The answers related to the other tasks had an accuracy between 11% and 23%, which is very low. However, accuracy increased using AwWidgets. Specifically, the partcipants gave answers with a high accuracy, between 83% and 100%. The unique case in which the accuracy is the same corresponds with Task 2 since Ubi4health notifies the emergency (the change) through striking and intrusive advice. Beyond this, the differences between the two versions of the application show a clear conclusion: participants had difficulty achieving the task when they had to control several aspects on different screens and manage many duties. These conditions made it difficult for the participants to compare the current state of a screen with a previous one. Accordingly, participants required ways to know what has changed in unattended displays. This information is provided by AwWidgets. AwWidgets allow users to be aware of what has happened on a display when they are not looking at it. Further, the user has to be able to continue with their current tasks once they have been informed of the change.

Some problems arose with people wearing glasses and using the version of Ubi4health with AwPanel. However, participants were able to answer the questions because AwPanel was in Book mode which implies that the widget stored the changes when the user was not looking at the corresponding screen. When the user looked at the screen again, AwPanel showed the changes in short periods of time due to eye detection problems. During these periods the participants were able to identify the changes but it took them longer since the book of changes appeared and disappeared from time to time.

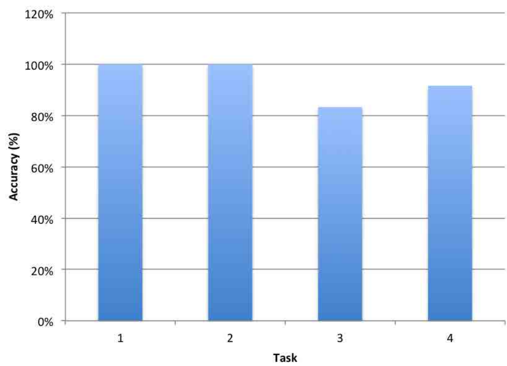

Figure 10: Average accuracy from the developers.

4.2 Developer's Point of View

In this evaluation we wanted to better understand developers fulfilled and unfulfilled wants and needs when using AwToolkit. We recruited 12 developers with a good knowledge of C# and expertise in using Visual Studio.

Every developer created an application that made use of AwWidgets, among other more standard user interface widgets. The developers had previously been introduced to AwToolkit via a simple tutorial that demonstrate its functionality and events. The tasks in the evaluation were:

- *Task* 1: import the library with AwToolkit to be able to see and use its items in the toolbar.
- *Task* 2: insert an AwButton and an AwPanel into the application and define how these widgets will behave when the user is not looking at the screen.
- *Task* 3: insert a 10-seconds timer to be run when AwButton is pressed.
- *Task* 4: disable the button after the 10-second timeout and introduce any change in the AwPanel.

The findings of the evaluations show that the use of AwToolkit is an easy-to-adapt process. Developers stated their satisfaction highlighting how similar it is to use compared to using the standard user interface controls. The developers' average accuracy was 100% on tasks 1 and 2, 83% on task 3 and 92% on task 4 (see Figure 10). This preliminary evaluation suggests AwToolkit can be straightforward to incorporate into the development process for a C# developer.

5. CONCLUSIONS

This paper has presented AwToolkit, a novel toolkit for developers that supports notifying users of changes that appear in the user interface when users are not looking at the display. The main objective is to allow users to manage complex systems in which the number of tasks to perform and the management of a large amount of data can overwhelm users. To solve this problem, AwToolkit offers the following attention-aware user interface widgets: AwPanel, AwButton, Aw Label, AwTextBox and AwComboBox. These widgets are based on standard user interface controls that contain customizable margins whose appearance changes based on the user's gaze and any changes detected. Gaze detection is performed using Diff Displays [3], a system that automatically identifies the user's gaze using computer vision techniques in conjunction with web cameras. AwToolkit extends this work by providing awareness of changes at the widget level. The widgets generate notifications with different levels of subtlety. Specifically, the levels mean that users are notified without being interrupted and having to stop their current task. In addition, one of the widgets allows users to be aware of changes in the user interface thanks to a book mode which enables user access to screenshots of each change that appears on a screen while the user is not looking at it. We performed an initial evaluation of AwToolkit with twelve end-users. The participants used Ubi4health [4], a healthcare application with a three-screen setting comprising a complex working environment. The results of the experiment have been positive. They have revealed the need for users to be aware of changes through subtle notifications. Otherwise, essential information could be unintentionally missed due to the complexity of these real-world tasks. AwToolkit has also been evaluated from the developers' point of view. Twelve developers implemented an application using AwWidgets. The evaluation with developers demonstrated that AwToolkit is easy to develop with due to its similarity with other standard user interface controls.

6. REFERENCES

[1] B. Arnrich, O. Mayora, J. Bardram, and G. TrÃűster. Pervasive healthcare, paving the way for a pervasive, user-centered and preventive healthcare model. *Methods of Information in Medicine*, 1:67–73, 2010.

[2] A. Bezerianos, P. Dragicevic, and R. Balakrishnan. Mnemonic rendering: An image-based approach for exposing hidden changes in dynamic displays. In *UIST 2006 Conference Proceedings*, pages 159–168, 2006.

[3] J. Dostal, P. Kristensson, and A. Quigley. Subtle gaze-dependent techniques for visualising display changes in multi-display environments. In *18th ACM International Conference on Intelligent User Interfaces (IUI 2013) Conference Proceedings*, pages 137–147, 2013.

[4] J. Garrido, V. Penichet, and M. Lozano. Ubi4health: Ubiquitous system to improve the management of healthcare activities. In *Pervasive 2012 Conference Proceedings*, 2012.

[5] C. Gutwin and S. Greenberg. Descriptive framework of workspace awareness for realtime groupware. *Computer Supported Cooperative Work*, 11:411–446, 2002.

[6] J. Hill and C. Gutwin. The MAUI toolkit: Groupware widgets for group awareness. *Computer Supported Cooperative Work*, 13:539–571, 2004.

[7] R. Madeira, O. Postolache, N. Correia, and O. Silva. Designing a pervasive healthcare assistive environment for the elderly. In *UbiComp 2010 Conference Proceedings*, 2010.

[8] M. Pichiliani and C. Hirata. Teleeye: An awareness widget for providing the focus of attention in collaborative editing systems. In *CollaborateCom 2008 Conference Proceedings*, pages 258–270, 2008.