WidgetLens: A System for Adaptive Content Magnification of Widgets

B. Agarwal, W. Stuerzlinger
York University, Toronto, Canada
http://www.cse.yorku.ca/~wolfgang

On displays with high pixel densities or on mobile devices and due to limitations in current graphical user interface toolkits, content can appear (too) small and be hard to interact with. We present WidgetLens, a novel adaptive widget magnification system, which improves access to and interaction with graphical user interfaces. It is designed for usage of unmodified applications on screens with high pixel densities, remote desktop scenarios, and may also address some situations with visual impairments. It includes a comprehensive set of adaptive magnification lenses for standard widgets, each adjusted to the properties of that type of widget. These lenses enable full interaction with content that appears too small. We also present several extensions.

Keywords: Magnification, semantic adaptation, windowing systems.

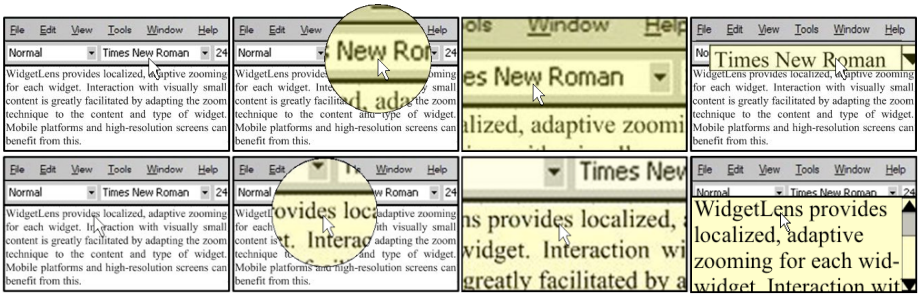

Fig. 1. *Illustration of different approaches to magnify screen content and their effect on the readability of the rest of the user interface. Top row: combo box, bottom: text canvas widget. From left to right: No magnification, circular lens magnification, whole screen magnification, and adaptive content magnification via WidgetLens.*

1 INTRODUCTION

Display resolutions are increasing faster than display sizes in computing. This leads to an overall increase in pixel density of screens, i.e. the number of pixels per unit space. The best place to observe this is smart phones and laptop screens. But even desktop monitors are now experiencing this trend. Higher density displays look more pleasing to the eye, can display more detail and seem easier to read. Also, computer users have often several applications running simultaneously, often sharing the whole area of the screen.

Pixel density is traditionally measured in pixels-per-inch (ppi) [27]. Current smart phones and portable book readers commonly feature screens between 150 and 400 ppi, laptops between 100 and 150 ppi, netbooks up to 200 dpi, and tablets between 130 and 265 dpi. Most high-end desktop monitors feature 100 ppi or more, with some, such as IBM's T221 display, up to 205 ppi. This increase in pixel density causes three problems.

First, most traditional desktop graphical user interfaces (GUIs), are designed for densities between 72 and 100 ppi. On screens with higher pixel densities, this makes all interactive user interface elements, the widgets, and all content appear smaller than intended. This issue is not only visible on current high-end laptops, but also by desktop users whenever they upgrade their monitor. Another indication is that the release of a 220 ppi laptop, the "Retina" Macbook, was in general very positively received. Yet, there widgets appear too small in software that has not been adapted to the high pixel density.

Second, remote access to desktop systems from mobile devices suffers from the fact that desktop GUIs are often designed for resolutions of 1280x1024 or higher. Yet, smart phones offer either less resolution or have much higher pixel

Proceedings of the 27th International BCS Human Computer Interaction Conference

densities. The situation is similar on tablets, which feature densities of 130 ppi or more. The current Apple iPad has 2048×1536 resolution at 264 ppi. Directly displaying a desktop remotely on such a device will result in content much smaller than intended, or force the user to use constant panning. The ubiquitous pinch-and-zoom functionality on mobile devices ameliorates the situation to some degree, at the cost of slowing users down a bit.

Lastly, humans with some types of vision impairment, such as color blindness and the growing percentage of older persons, may also benefit from mechanisms that magnify and adapt screen content. The general population suffers from similar issues in outside scenarios with bright light or on devices with high-density screens.

Screen magnifiers are the most common solution to the problem of content appearing too small. Most operating systems provide them. Either a fixed portion of the screen is zoomed inside a lens or the whole screen is magnified and panned whenever the cursor hits the edges of the screen. Yet, a fixed size lens magnifies only a limited region of the screen, which works only for widgets fitting within that region. Panning has the downside that the user may lose their overview of the screen. Long text fields, large lists or tables, and canvas regions are examples where neither of these two strategies works well. Figure 1 illustrates these problems.

For canvas regions, one can address the problem of content appearing too small with an appropriate zoom factor – as long as the application provides said zoom functionality. Yet, zooming is limited to the main content area(s), and does not extend to the widgets, toolbars, secondary dialogs, and other GUI components. For all such standard widgets, there is currently no solution, as all current major GUI toolkits do not support seamless scaling of widgets. In other words, there is no API support for automatic resizing of GUI elements or different pixel-density dependent rendering of GUIs. An exception is the Apple iOS platform for smartphones and tablets. iOS can scale content through pixel doubling in both dimensions. Alternatively, it also permits the programmer to provide two versions, adapted to the two different pixel densities. For all other platforms, it is difficult for programmers to create GUIs that automatically adapt to a given pixel density, especially for non-integral size factors. Unfortunately, we see no concerted effort to create GUI toolkits that support seamlessly resizable widgets with no or only low programming overhead.

Beyond the above-mentioned issues around small content, a central issue is that interaction with GUIs displayed on screens with higher pixels densities is also difficult. First, users need to hit (very) small targets. Several techniques have been proposed to address this problem and we review them below. Second and more importantly, the user needs to be able to interact with the displayed content. This includes recognizing the widget, clicking on subparts of it, such as a “drop down” arrow or scrollbars, cursor positioning, and text. On displays with higher pixel densities than those used to design the GUI all of these interactions demand better motor control and vision relative to the norm. Especially older adults find this challenging.

1.1 Previous Work

In this section we first survey techniques to facilitate small target acquisition and then give an overview of various techniques to adapt content.

Target acquisition has been improved through modifying the presentation of targets or the cursor. The first approach modifies the virtual size of a target [46] and occasionally brings the target closer (Drag-and-pop [4]). The latter replaces the point cursor with a situationally ‘adapted’ cursor. This includes shrinking and enlarging activation areas [22, 31, 46], variable Control-Display (CD) gain ratio cursors [1, 6, 25], as well as magnetic cursors, Object Pointing [23], and Predictive Interaction [3]. These technique target near optimal pointing performances and assist also people with visual and motor impairments for pointing tasks.

Lens based methods magnify a part of the screen content inside a non-interactive lens. Such lenses are available in all windowing systems. The idea of copying regions of a screen and showing them in a new separate window, either as a control or an overlay, is related. WinCuts [41] is a system that copies arbitrary window regions and shows these inside read-only “mini-windows”. Ramos et al. [37] presented pressure-based pointing lenses that magnify a 128x128 pixel window. Another approach is exemplified by the Apple iPhone text-editing lens and Shift technique [44], which uses a ‘callout’ to show a copy of the occluded screen area. This lens pops up when the user performs a touch and hold gesture. This delay-activated lens is used only for cursor positioning. The Pointing Magnifier [30], an extension of the area cursor [46], is a visual motor magnifier. For widgets larger than the magnification area or widgets that expand on interaction, such as a menu, this strategy fails, as the user is then not able to interact directly with the part of the widget outside the lens. TapTap [38] is a thumb-based interaction method to improve accessibility at screen corners. It uses a double-tap gesture, where the first tap shows a magnified lens for the selected region. The second tap then selects the target in that region and closes the lens.

A common issue with lens-based approaches is that the lens itself occludes neighbouring content, either completely or partially. This may lead to (partial) loss of context. The Elastic Presentation Framework [7] presented various distortion lenses.

Other examples for focus-and-context techniques are fisheye [24], Sigma [36], and high-precision magnification [2] lenses. Rapid transition between focus and context is important for the efficient use of such lenses. Pietriga [36] noted that in general small lenses do not occlude neighbors and thus offer more screen real estate to the context. Consequently, they are more efficient.

There has been substantial work on interface adaptation [15, 17, 20, 21, 40]. These user initiated customization methods improve user experience and enable otherwise impossible tasks. Supple reorganizes content hierarchies based on input methods [20] and user abilities [21]. Similarly, Prefab [15] addresses language difficulties by providing translation. Bubbling Menus [43] is targeted at expert users and uses directional hints for activation. Findlater et al. [18, 19] stated that adaptive menus have a very positive impact on performance and satisfaction on small devices. Findlater et al. [17] presented ephemeral adaptation of a menu widget, and demonstrated benefits in visually complex tasks. The implementation of adaptive user interfaces on top of existing, non modifiable, systems requires GUI interpretation through accessibility APIs or pixel-based GUI recognition. All major GUI toolkits support accessibility APIs. Pixel-based methods, such as Sikuli [47] and Prefab [15], work only on the visible parts of a GUI. Hybrid approaches such as PAX [9] and Deep Shot [8] combine pixel- and accessibility API-based methods.

Toolglasses and MagicLenses [5] is a see-through interface that modifies presentation of objects seen through them. UI Façades [40] extends the Metisse windowing system [11]. This system can adapt, re-configure, and re-combine existing GUIs and even replace widgets. Supple++ [21] presented ability-based interface rendering, where the content is varied based on the user's abilities.

Finally, there are a few isolated examples of adaptive widget magnification on current platforms. The Apple iOS Safari browser magnifies HTML lists with an enlarged selection widget, which takes over a large part of the screen. However, this is limited to one or two kinds of widgets and is not sufficient to handle the whole GUI of an application. The only other system that provides facilities to magnify widgets for GUI applications is Scotty [16]. Yet, this system uses the print functionality to achieve magnification, which is very resource intensive. Also and as presented, this system does not automatically magnify widgets on demand.

2 THE WIDGETLENS SYSTEM

We present a new system, WidgetLens, designed to enable easy interaction with GUIs displayed on high pixel density screens. Simultaneously, it may also improve accessibility for users with degraded vision. We present the following contributions:

- A system that implements new, on-demand, localized, widget-dependent, and automatic zoom lenses, implemented without altering the underlying applications or GUI toolkits
- Widget magnification techniques that enable full interaction with magnified content for all standard GUI widgets.

This novel solution also bridges the gap between screens with increased pixel densities and the fact that GUIs today are still mostly resolution dependent. WidgetLens improves the "look" of interfaces on high-pixel-density screens, and also assists user interaction, i.e. improves the "feel".

2.1 Design Decisions

We made the following main design decisions, based on results from some previous work, cited below, and our own exploratory studies:

- **Location:** A WidgetLens is shown directly (centered) over the original widget. This avoids obscuring other, potentially relevant, content as much as possible.
- **Size:** If possible, the system overlays a WidgetLens within the original widget area. This is viable for some widgets, as margins are often generous and horizontal scrolling is also supported. For widgets where a WidgetLens cannot fit within the original, the magnification factor is used.
- **Magnification Factor:** Each WidgetLens magnifies the content of the original widget by a global, user configurable, zoom factor. A factor of 2x is sufficient for most cases except for displays with more than 200 ppi.
- **Visual Appearance:** To highlight the lens, each WidgetLens is blended with a pastel shade, by default a *pale yellow*. However, users can change this colors setting to suit individual needs (see Figure 3).
- **Interface Interpretation:** The WidgetLens system relies on accessibility information retrieved from the underlying GUI toolkit. Alternatively, the system could be built on pixel-reversal techniques [15, 8] or, better, hybrid approaches [9]. While WidgetLens provides high-resolution lenses for all standard widgets, non-standard ones are handled with texture scaling. Developers can also supply a custom script instead.
- **Activation:** A central problem for lens-based interaction techniques is that one cannot know with certainty in what situation the user wants to pop up or destroy the lens. There are multiple options to activate a lens. These include pressure and delayed

activation, context-aware activation via "hover" events or cursor speed, or modifier key/button based activation. Similar to the delay lens [37], WidgetLens pops up a lens centered over the current widget when the cursor is stationary for more than a half a second. The appearance and disappearance of a WidgetLens is animated.

- **Layout Changes vs. Overlays:** In simple icon magnification systems, such as the Mac OS X dock, other widgets *move* to accommodate the overall increase in size. In general, this is not possible for densely tiled two-dimensional layouts, such as most GUIs. While it might be theoretically possible to use layout managers as a fix, this approach suffers from several issues. First, it is currently not possible to access that layout manager from the outside. Second, expanding a single widget will negatively impact the overall layout and may only be possible by growing the window. Finally, some GUI windows do not support resizing. The WidgetLens system thus uses overlay windows instead.
- **Interaction Adaptation:** Depending on the type of each given widget, the interaction inside a WidgetLens is adapted to facilitate interaction. E.g., for left handed users, the scroll bar appears at the left side. Also, scroll bars are displayed only if there are too many entries in the magnified lens.
- **Deactivation:** WidgetLenses for all simple widgets, such as icons, buttons, combo-boxes, and single-selection lists, are deactivated whenever any type of selection event occurs. For other widgets that permit complex interactions, such as a multi-line text area, the user has to move the cursor outside of the lens to destroy it.

One of the primary objectives of the WidgetLens system is to show and enable users to interact with magnified content. This offers a three advantages. First and even on screens with high pixel densities, users can still use screen space as usual with multiple windows. Second, users can use this method to remotely access desktop applications on mobile devices with low-resolution screens. Third, WidgetLens also may give a subset of those users with moderately degraded vision better access to GUIs and may enable them to interact fully with standard GUI applications, without forcing them to resort to traditional accessibility-based solutions.

2.2 WidgetLenses

The WidgetLens system uses three techniques to improve user interaction: *automatic widget replacement, semantic adaptation,* and *on-demand presentation*. Each WidgetLens uses these.

Automatic Widget Replacement. UI Façades [40] presented *manual* widget replacement before. The WidgetLens system *automatically* adapts widgets by presenting a widget's content in an appropriately magnified way. Interaction with the replacement widget is mapped to corresponding events on the original one.

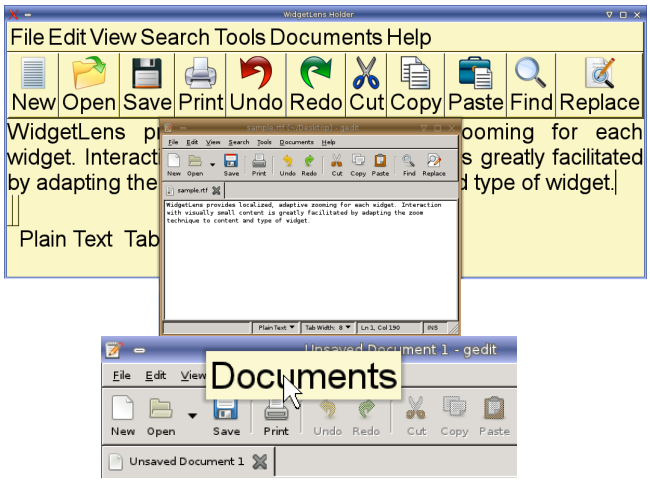

Fig. 2. *Top: WidgetLens holder with 4x magnified widgets (in the back) and original application window. Bottom: hovering over a menu widget shows its high-resolution WidgetLens. For simplicit of illustration, only the WidgetLens of the focussed widget is shown.*

Semantic Adaptation. For each new window, all its widgets are re-created and packed one-to-one in a separate, hidden window, called WidgetLens holder (Figure 2). For this, we walk the widget tree recursively and add a *semantically adapted* version of each widget based on its accessibility information. Widgets are adapted both in size and content. For example, text areas use larger font sizes and resort to multiline text when the number of characters exceeds the allotted space. For icons, intelligent scaling or a higher resolution icon library is used. Examples are shown throughout the paper.

On-Demand Presentation. At runtime, each widget lens is *presented* in an overlay *on demand* whenever the cursor hovers over the original widget for more than a half a second (or alternatively is activated through a "long" click). This is another innovation relative to UI Façades. See the bottom part of Figure 2 for an example of the result of a hover action. While activated, all mouse and keyboard events on the widget lens are absorbed by the replacement and are passed to the application through event redirection via accessibility callbacks (or through coordinate translation for texture-scaled widgets). For this, all widget events on a replacement widget are intercepted and communicated back to the original application in an appropriate manner. This enables normal user interaction with the content of each WidgetLens, while still ensuring that the interaction is reflected in the application itself.

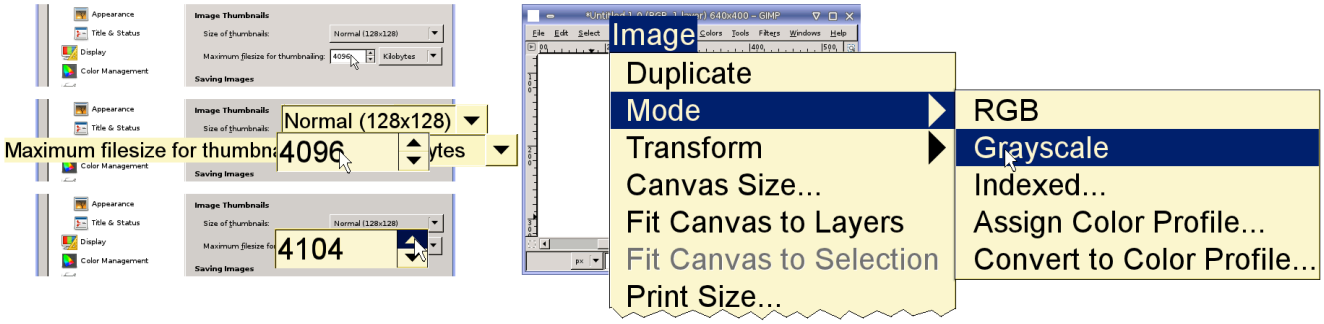

Fig. 3. *High-resolution WidgetLenses for a spin button and a menu. Note the absence of pixilation artifacts and the larger fonts. Left: WidgetLenses for the main widget (4x) and its neighbors (2x) appear if the cursor remains still (middle left). With the first click or key event on the lens, all neighboring lenses disappear and appropriate callbacks are issued to the application (bottom left). Right: Magnified menu cascade in the Gimp image editor (cut to conserve space).*

3 USER INTERACTION AND ADAPTATIONS

When the cursor enters the area of a widget in the normal application, the WidgetLens system displays the corresponding adapted WidgetLens from the WidgetLens holder. The WidgetLenses for the immediate neighbors are displayed around the focused widget in a lower layer, see Figure 3. The user then interacts with the top WidgetLens as with normal application widgets. The results of any interaction results or changes to the content of the WidgetLens are forwarded to the underlying application to make the system appear seamless.

All the functionalities discussed in the following also extend the widget replacement facilities of the underlying UI Façades system further.

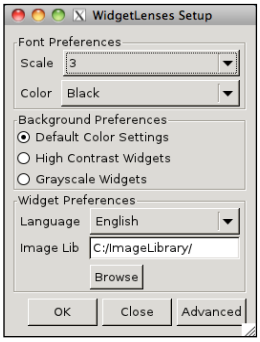

Fig. 4. *Control panel for user preferences to express situational needs.*

3.1 User Specific Adaptations

A user can express his or her situational needs and preferences for the current application session within WidgetLens (Figure 4). One benefit of this is that the user is not forced to make system wide changes. Instead, only the applications running under the control of the WidgetLens system are affected. The adaptation options available cover many of the vision and language related difficulties that the general population may face during GUI interaction. Each of these is discussed in turn.

Users can choose a global magnification factor for widget and image scaling as well as a font. All the lenses then use the specified font for their content. Additionally, users can provide a higher resolution replacement image library for widgets that contain icons, such as buttons and menu-items. If no high-resolution icons are available, the WidgetLens system uses HQx [26], an image-scaling algorithm targeted at pixel art, to create larger versions of each icon. While not perfect, the results are significantly better compared to the blurry results of naïve image up-scaling or the blocky appearance of pixel doubling, see Figure 5. This ensures that all image content remains as readable as possible for the chosen magnification factor. There are other approaches for scaling pixel art, e.g., [32].

Fig. 5. *Low-resolution icons (small, left column) scaled using naïve texture-scaling (second column), image up-scaling (third column) and HQx respectively (right).*

WidgetLens implements several accessibility features via color settings. The user can customize WidgetLens to show high-contrast lenses based on appropriate standards for low-vision users [29]. WidgetLens then invert all colors in the WidgetLenses. This is designed to help users from the general population in bright outside scenarios, but may also help people with vision impairments. Color-blind users, a common type of vision impairment, can also select an appropriate adaptation. This operation then enables appropriate color choices and image filters. See Figure 6 and Figure 7.

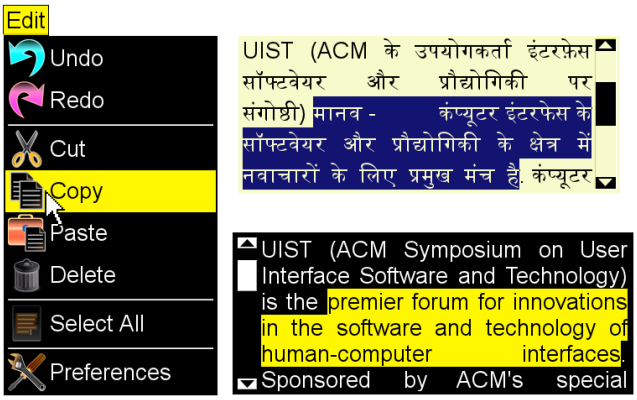

Fig. 6. *High Contrast WidgetLenses for different visual deficiencies. Left: High contrast version of a menu, which inverts the colors of all content. Right: Text areas in Hindi and English. The bottom WidgetLens is further adapted for left handed use.*

Moreover, WidgetLens also supports left-handed users by moving scroll bars and similar interactive parts of widgets, such as the triangle for a combo-box, to the left side. See Figure 6 for an example.

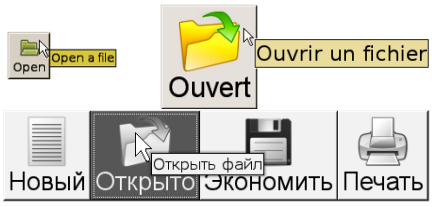

Fig. 7. *Top: Illustration of high-resolution WidgetLens adapted to French. All widget text and tooltips are adapted in size and language. Bottom: Part of G-Edit toolbar adapted to gray scale and to the Russian language.*

Similar to Prefab [15], the WidgetLens system addresses language difficulties by optionally providing widget content in a user preferred language. For example, a user can specify 'French' as his preferred language and will then see all WidgetLenses in that language (Figure 7). For this, all widget text is translated to the specified language using an online web service [28].

3.2 Widget Specific Adaptations

The size and shape of a WidgetLens is magnified in proportion to the corresponding original application widget. However, WidgetLenses generated with this straightforward method are sometimes too large for practical use and thus the system needs to adapt them further. One common example is text fields that already span most of the screen width, such as the address field in a web browser. Another example are composite widgets, such as tables or trees, which may have free space inside or around them that can be used to improve visual presentation. The WidgetLens system deals with such widgets in a series of steps. First, it prioritizes the area taken up by a widget's content, such as text and images, over free space during the widget duplication step. For this, the dimensions of a WidgetLens are calculated as the minimum of the size of the original widget and the actual ones of the magnified content. This ensures that a WidgetLens is appropriate for the content and size of the current widget while minimizing white space. E.g., if a list widget has space for ten entries, but contains only five items, only those five items are magnified. Second, if a WidgetLens is still too large to fit onto the screen, we limit the number of text lines, list or tree items, or the number of characters that are displayed within a WidgetLens to the available space (see Figure 8).

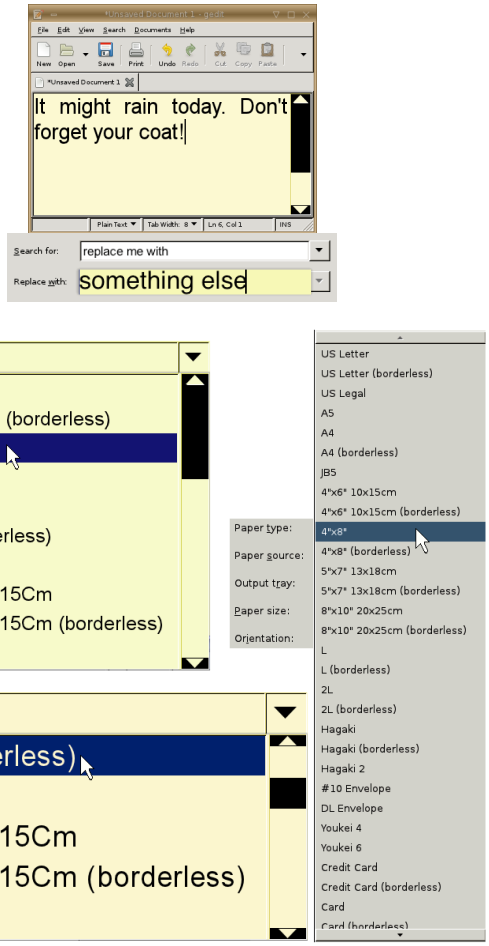

Fig. 8. *Top: WidgetLenses for Text-areas and field may fit inside the original widget boundary. Bottom-left: Several combo-box WidgetLenses (2x and 3x), shown for different simulated pixel densities.*

4 IMPLEMENTATION

The WidgetLens system is built on top of Metisse [11] and User Interface Façades [40]. Metisse provides all basic window manager functionality, such as event registry, event propagation, window stacking and positioning. UI Façades is used to show WidgetLenses and to enable user interaction

with them. All content duplication, magnification, interaction, and event adaptation functionality is unique to the WidgetLens system, as are several low-level extensions to Metisse and UI Façades, such as WidgetLens focus management, overlay lens activation and disappearance, texture scaling, and some low-level event redirection. Figure 9 illustrates the high-level structure and event flow of the WidgetLens system.

The extensions to Metisse (and UI Façades) are implemented in C/C++. One noteworthy issue here is that WidgetLens needs to work around the fact that Metisse or UI Façades does not expose individual façades to other parts of the system. All high-level WidgetLens functionality, such as the various types of WidgetLenses and their event callbacks, is implemented as Python scripts on top of UI Façades. Accessibility APIs are used to communicate with the application's widgets. Alternatively, this could have been implemented with similar technologies [8, 9, 16].

When the cursor enters the area of a widget in the original application, the WidgetLens system identifies and displays the corresponding magnified WidgetLens from the WidgetLens holder. The WidgetLenses for the immediate neighbours are handled similarly and displayed around the focused widget in a lower layer. The user then interacts with the top lens as with normal application widgets.

WidgetLens provides magnification for all standard widgets, such as buttons, toolbars, menus, text fields & areas, combo-boxes, lists, and tables. The only standard widget that is not handled in the WidgetLens system is canvas areas. The main reason for this is that most applications, such as editors and browsers, already implement a zoom facility for their canvas regions. Custom or non-standard widgets, such as a color picker area, are handled in WidgetLens through texture scaling by default. Alternatively, a developer can also provide a custom Python script to enable WidgetLens to magnify their widget.

Simple selection events are directly forwarded to the application and appropriate visual changes are shown in the WidgetLens. This ensures that any interaction with simple widgets, such as buttons, icons, and menus, is immediately communicated to the application. For WidgetLenses that are *texture-scaled*, event coordinates are translated back to the original widget. Interactions that directly affect the content, such as character insertion, are also communicated immediately to the application. Interactions that do not affect the content directly, such as dropping down a menu or scrolling a text field, are handled within the WidgetLens. Upon disappearance of a WidgetLens the final content of the widget is again synchronized with the running application to ensure consistency. This effectively implements indirect interaction through the magnified replacement widget seamlessly.

To illustrate some of the intricacies of this indirection, consider interaction with a menu bar. The first click on a top-level menu item is captured and handled by the WidgetLens system to identify the submenu. At this point, the high-resolution version of both the selected menu-item and its corresponding dropdown menu are visible. A click on a submenu item then results in a menu selection action and destroys the WidgetLens. However, if this is another sub-menu, the second event also gets absorbed by the WidgetLens and shows the sub-sub-menu, and so on. See Figure 4. Combo-boxes and other widgets that change appearance upon the first interaction are handled similarly.

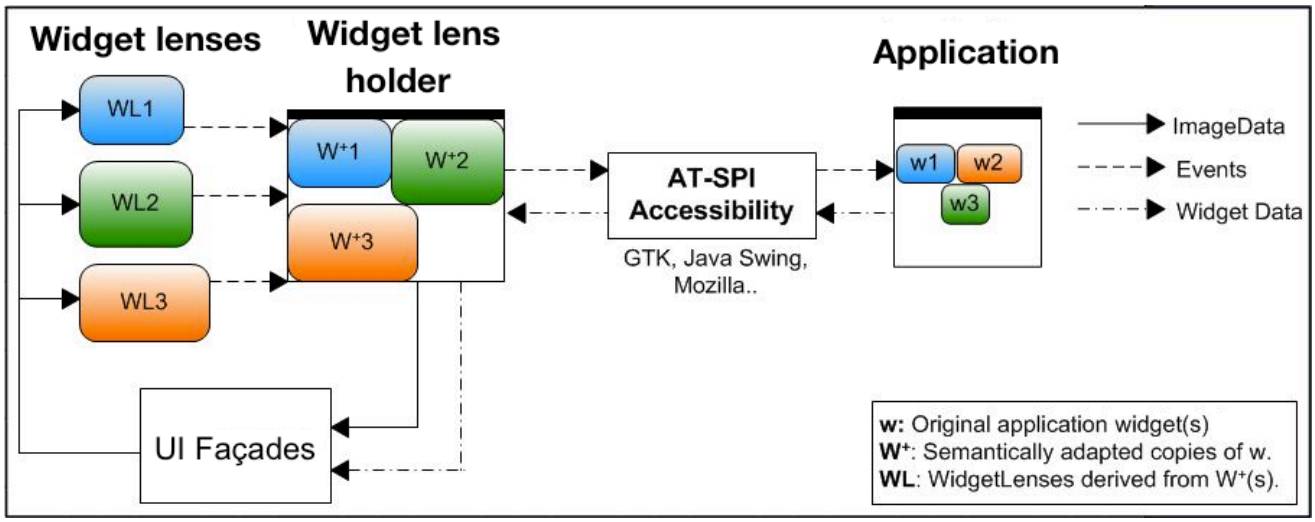

Fig. 9. *Event and data flow for the WidgetLens system. Widget data is retrieved from the application via the accessibility API. The system creates an enlaged version for each widget in the WidgetLens holder, invisible to the user. WidgetLens shows parts of the WidgetLens holder with the help of UI Façades as an overlaid lens during run-time on demand. Events a WidgetLens are passed back to the original application through the accessibility API. Note that both UI Façades and Metisse had to be extended for WidgetLens. Metisse is the underlying window manager and not shown for simplicity.*

All user interactions with a text field or area widget, such as insert, delete, highlight or erase events, are handled by the WidgetLens system. For cut, copy, and paste events, WidgetLens creates appropriate callbacks to the application. Drag and drop is also supported if the user first performs a long mouse button press to activate a WidgetLens, then dragging the (magnified) content, and finally dropping it at the desired location.

To address interactions with potential side effects, such as the user pressing a shortcut key for an operation that also changes the state of another widget, the WidgetLens system monitors for changes in other widgets of the original application. It updates the WidgetLens holder accordingly whenever such a change occurs.

Replacement widgets are only generated upon application initialization and whenever a new window is created. Window resizing or opening and closing a view inside a window is a noteworthy exception here, as all resizable widgets may change size then. To address this, the WidgetLens system monitors the running application for widget size changes in two ways: First, we check for a resize event on the original application window. Second and at regular intervals (by default a second), the size of each widget is compared with the last seen size. This handles cases where the application resizes widgets without a change in the window size. Regardless of the cause, that change is then reflected in a corresponding size change for the corresponding WidgetLenses.

5 DISCUSSION & FUTURE WORK

WidgetLens targets GUI accessibility on displays with high pixel densities, mobile scenarios, and may also address issues encountered by people with some forms of visual impairments. While widget magnification may seem like a transitional technology, we point out that traditional GUIs will always show up too small on mobile devices, simply due to the substantial differences in screen size. Moreover, WidgetLens is particularly suited for mobile scenarios where the whole screen is transmitted scaled down to the target resolution (to reduce bandwidth consumption). In this situation, WidgetLens would show the magnified widgets at a larger size, more appropriate for interaction. Finally, and If part of WidgetLens were implemented locally on the tablet, it would then theoretically suffice to send only the type and content of the currently magnified widget, which requires little overhead.

In our experience and in an exploratory study with students on a “Retina” MacBook, the system achieves its design target and enables seamless interaction with unenhanced applications, where all widgets appear too small. We found only minor implementation issues. However, we caution that we have not yet formally evaluated WidgetLens with outside users or with people with vision impairments. We plan to do this in the future. In the following, we discuss some options that we have not yet explored, but may also tackle in the future.

5.1 Alternate Lens Activation and Deactivation

The “best” method for activation and deactivation of a WidgetLens is an interesting topic. Our current implementation uses delayed activation. This context-based technique essentially leaves the job of activating and deactivating a WidgetLens to the system. The main benefit is that the user never needs to perform an explicit action to request a lens, which reduces activation overhead. With delay-based activation the system uses the cursor position and velocity to decides when to activate and deactivate a WidgetLens. Thus, when the cursor is immobile for more than half a second over a widget the system shows a WidgetLens.

Another approach is to activate a WidgetLens based on the user’s explicit request. A few options here are context menus, function or modifier keys, and various gestures. Although this approach gives more control to the user, it also increases interaction overhead through additional actions, such as a context menu activation and selection. Also, a context menu itself needs to be already a high-resolution WidgetLens to provide adequate accessibility. Modifier keys require a “free” key in the windowing system. However, many systems have already assigned all likely candidates. Also, mouse gestures are not widely accepted, see e.g., the statistics on gesture usage in browsers.

Another approach is click-based activation. Here, a click on the unmagnified widget shows the WidgetLens, similar to the pointing magnifier [30]. For simple widgets this introduces extra interaction overhead. Consider that a WidgetLens for a button then effectively requires two clicks to achieve a single action: one to activate the lens, the other to click on the button, which also destroys the lens as well. However, for some widgets, such as a combo-box or menu, this first click can be re-used. Thus, that first click can both activate a WidgetLens and drop the list of options down. Hence, there is no real overhead for this kind of widget.

5.2 Adding Motor Magnification

Visual enlargement of a target improves readability but does not necessarily afford better user interaction performance. Given that interaction with un-scaled widgets is hard on screens with high pixel densities, it may make sense to magnify the motor space as well. Note that while an un-scaled widget itself is often still large enough to hit on such a screen, components of it, such as the little up/down arrows in “spinner” widgets, may be too

small to interact with. This is documented by the complaints about traditional GUIs on high-pixel density screens. Hence, we propose high-resolution widgets that also offer motor magnification to address this issue.

5.3 WidgetLens Integration into GUI Toolkits

The WidgetLens system is best integrated within the windowing system or GUI toolkits, rather than built on top of a modified window manager and UI Façades. However, implementing WidgetLens inside GUI toolkits requires substantial work. Hence, we discuss here another, potentially more viable, approach that integrates different modules of the WidgetLens system at different levels. In this approach, window management support for real-time copying of screen regions and the associated event redirection is implemented at the windowing or operating system level. Scalable widget support is implemented at the toolkit level, together with the necessary interaction event hooks. The toolkit can then witness user interaction events first-hand and manage the real-time widget copies, based on these events. A new API offered by the toolkit can then be used for custom widgets and their WidgetLenses. Defining appropriate APIs for this is an interesting area for future work.

Finally and as our system is targeted at end users, we do not expose internal options and parameters to the user. For power users it might be appropriate to expose more control, e.g., over the word wrap option for magnified text.

5.4 Advanced White Space Management

Tables are handled on a single-cell basis with column-based magnification. An advanced solution would offer row-wise magnification as an alternative. This could be handled as follows. If there is free white space available within a column, said space could be used to display the magnified content. Otherwise, entries would be clipped to fit in the available space and the full high-resolution content only shown in a tooltip. This is similar to how Microsoft Excel handles the problem.

6 CONCLUSION

We presented WidgetLens, a system that provides adaptive widget magnification to improve access to and interaction with magnified graphical user interfaces. WidgetLens shows magnified versions of all standard widgets on demand. Each of these WidgetLenses provides full interaction with the content, adjusted to the properties of the particular type of widget. The system is targeted at usage on displays with high pixel densities, where traditional content can be too small for easy interaction, at mobile scenarios where high-resolution content is shown on small screens, and also at some scenarios where vision impairments play a role.

In the future, we plan to extend WidgetLens into some of the mentioned directions. With funding, we will also evaluate WidgetLens in a formal user study on screens with high pixel densities with older adults to verify its effect on interaction performance. At a global level, we see WidgetLens as an inspiration to the field to think about better ways to adapt traditional GUIs to today's technologies and use cases.

REFERENCES

1. Ahlström, D., Hitz, M. Leitner, G. An Evaluation of Sticky and Force Enhanced Targets in Multi Target Situations. *NordiCHI 2006*, 58-67
2. Appert, C., Chapuis, O., Pietriga, E. High-precision magnification lenses. *CHI 2010*, 273-282.
3. Asano, T., et al. Predictive interaction using the delphian desktop. *UIST 2005*, 133–141.
4. Baudisch, P., et al. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch- and pen-operated systems. *INTERACT* 2003, 57-64.
5. Bier, E., et al. Toolglass and magic lenses: the see-through interface. *SIGGRAPH 1993*, 73-80.
6. Blanch, R., et al., Semantic pointing: improving target acquisition with control display ratio adaptation. *CHI 2004*. 519–526.
7. Carpendale, M.S.T., Montagnese, C. A. Framework for Unifying Presentation Space. *UIST 2001*. 61-70.
8. Chang, T. and Li, Y. Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras. *CHI 2011*, 2163-2172.
9. Chang, T., Yeh, T., and Miller, M. Associating the Visual Representation of User Interfaces with their Internal Structures and Metadata. *UIST 2011*. 245-256.
10. Chapuis, O., Labrune, J., and Pietriga, E. DynaSpot: Speed-dependent area cursor. *CHI 2009*, 1391-1400.
11. Chapuis, O. and Roussel, N. Metisse is not 3D desktop! *UIST 2005*, 13–22.
12. Cockburn, A. and Brock, P. Human on-line response to visual and motor target expansion. *GI 2006*, 81-87.
13. Cockburn, A. and Firth, A. Improving the acquisition of small targets. *HCI 2003*, 181–196.
14. Cockburn, A., Karlson, A. Bederson, B. A review of overview+detail, zooming, and focus+context interfaces. *CSUR 2008*, 41(1):1–31.

15. Dixon, M. and Fogarty, J. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *CHI 2010*, 1525-1534.
16. Eagan, J., Beaudouin-Lafon, M., Mackay, W. Cracking the cocoa nut: user interface programming at runtime, *UIST 2011*, 225-234.
17. Findlater, L., et al. Enhanced Area Cursors: Reducing fine pointing demands for people with motor impairments. *UIST 2010*.
18. Findlater, L., McGrenere, J. A comparison of static, adaptive, and adaptable menus. *CHI 2004*, 89-96.
19. Findlater, L. and McGrenere, J. Impact of Screen Size on Performance, Awareness, and User Satisfaction With Adaptive Graphical User Interfaces. Proc. CHI 2008, 1247-1256.
20. Gajos, K. and Weld, D. S. SUPPLE: automatically generating user interfaces. *IUI 2004*. 93-100.
21. Gajos, K., Weld, D. S. and Wobbrock, J. O. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence 2010*, 910-950.
22. Grossman, T. and Balakrishnan, R. The Bubble Cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. *CHI 2005*, 281-290.
23. Guiard, Y., Blanch, R., and Beaudouin-Lafon, M. Object pointing: a complement to bitmap pointing in GUIs. *Graphics Interface 2004*. 9-16.
24. Gutwin, C. Improving focus targeting in interactive fish- eye views. *CHI 2002*, 267–274.
25. Hourcade, J. P., et al. Pointassist for older adults: analyzing sub-movement characteristics to aid in pointing tasks. *CHI 2010*. 1115-1124.
26. http://en.wikipedia.org/wiki/Hqx
27. http://en.wikipedia.org/wiki/List_of_displays_by_pixel_density
28. http://*www.translator.google.com*
29. http://www.cnib.ca
30. Jansen, A., Findlater, L., and Wobbrock, J. O. From the lab to the world: Lessons from extending a pointing technique for real-world use. *Extended Abstracts CHI 2011,* 1867-1872.
31. Kabbash, P. & Buxton, W. The Prince" technique: Fitts' law and selection using area cursors. *CHI 1995*, 273-279.
32. Kopf, J., and Lischinski, D. Depixelizing pixel art. *SIGGRAPH 2011*, 99:1-99:8.
33. McGuffin, M. and Balakrishnan, R. Acquisition of expanding targets. *SIGCHI 2002*, 57-64.
34. McGuffin, M. and Balakrishnan, R. Fitts' law and expanding targets: Experimental studies and designs for user interfaces. *CHI 2005*, 388–422.
35. Peck, S., North, C., and Bowman, D. A multi-scale interaction technique for large, high-resolution displays. *3DUI 2009*, 31-38.
36. Pietriga, E., and Appert, C. Sigma lenses: focus-context transitions combining space, time and translucence. *CHI 2008*, 1343-1352.
37. Ramos, G., et al. Pointing lenses: facilitating stylus input through visual-and motor-space magnification. *CHI 2007*. 757-766.
38. Roudaut, A., Huot, S., Lecolinet. E. TapTap and MagStick: Improving one-handed target acquisition on small touch-screens. *AVI 2008*. 146-153.
39. Ruiz, J. and Lank, E. Speed pointing in tiled widgets: understanding the effects of target expansion and misprediction. *Intelligent User Interfaces 2010*, 229-238.
40. Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*, 309-318.
41. Tan, D., S., Meyers, B., and Czerwinski, M. WinCuts: manipulating arbitrary window regions for more effective use of screen space. *CHI 2004*. 1525-1528.
42. Taras, C., et al. Improving screen magnification using the HyperBraille multiview windowing technique. *ICCHP 2010*, 506-512.
43. Tsandilas, T., Schraefel, M.C. Bubbling menus: a selective mechanism for accessing hierar-chical dropdown menus. *CHI 2007*. 1195–1204.
44. Vogel, D. and Baudisch, P. Shift: A Technique for Operating Pen-Based Interfaces Using Touch. *CHI* 2007.
45. Wobbrock, J., et al. The angle mouse: target-agnostic dynamic gain adjustment based on angular deviation. *CHI 2009*, 1401-1410.
46. Worden, A., et al. Making computers easier for older adults to use: area cursors and sticky icons. *CHI 1997*, 266-271.
47. Yeh, T., Chang, T. and Miller, R. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009*, 183-192.1.