Visual Code Widgets for Marker-Based Interaction

Michael Rohs
Institute for Pervasive Computing
Department of Computer Science
ETH Zurich, Switzerland
rohs@inf.ethz.ch

Abstract

We present a set of graphical user interface elements, called widgets, for 2-dimensional visual codes. The proposed widgets are suitable for printing on paper as well as showing on electronic displays. Visual code markers and their orientation parameters are recognizable by camera-equipped mobile devices in real time in the live camera image. The associated widgets are specifically designed for marker-based interaction. They define basic building blocks for creating applications that incorporate mobile devices as well as resources in the user's environment, such as paper documents, posters, and public electronic displays. In particular, we present visual code menus (vertical menus and pie menus), check boxes, radio buttons, sliders, dials, and free-form input widgets. We describe the associated interaction idioms and outline potential application areas.

Figure 1. A user interacting with a printed visual code pie menu. On the phone screen, the focused pie slice has a yellow frame.

1. Introduction

Widgets are generic, reusable, directly manipulable, self-contained visual screen idioms. They are the primary building blocks for creating graphical user interfaces. Examples are buttons, check boxes, edit fields, and tooltips. Operating systems typically come with a default widget toolkit, which defines a set of basic interaction metaphors across all applications. Each widget solves a certain input or interaction problem and offers familiar affordances to the user. Cooper [4] classifies widgets into the following four categories: *Imperative widgets*, such as buttons, initiate a function. *Selection widgets*, such as check boxes or radio buttons, allow the user to select some option or data. *Entry widgets*, which can either be bounded, like sliders, or unbounded, like edit fields, enable the input of data. *Display widgets*, which will not be discussed here, serve the manipulation of the appearance of an application itself.

In this paper, we present a number of widgets that are specifically designed for 2-dimensional visual markers, called *visual codes* [12]. Visual codes can be recognized by camera-equipped mobile devices, such as PDAs and mobile phones, in real time in the live camera image. Recognition includes a number of orientation parameters that describe the orientation of the device relative to the code. It thus becomes possible to define visual code equipped widgets and design graphical user interfaces that can literally be printed on paper or shown on large-scale displays [1].

With the use of such widgets the expressivity of marker-based input becomes richer, which enhances the overall user interface capabilities of mobile devices. In this way, printed documents and large-scale displays serve as extended user interfaces of mobile devices, which are not subject to the size restrictions of handheld device screens. Moreover, paper is permanently available, always ready for interaction,

and – when used with visual code widgets – can help to quickly establish the context of an application.

Interaction would typically take place as follows (see Figure 1): The user finds a visual code widget, for example in a magazine, on a doorplate, on a poster, or on a survey form. She starts the generic recognizer application on her phone or PDA and aims at the widget. The widget appears on the device screen in view finder mode and is updated in real time as the user moves the device relative to the widget. The state of the widget is superimposed over the camera image, for example by drawing a current slider position. The user knows how to interact with the widget, since the familiar layout of the widget offers clear affordances. It allows her to quickly enter data or to make selections. The context of the interaction is implicitly established and takes almost no time or effort for the user to set up.

In the following section, we discuss related work. Section 3 provides an overview of visual code features. Section 4 presents visual code menus and Section 5 describes selection and data entry widgets. Section 6 outlines application areas and a conclusion is given in Section 7.

2. Related Work

In comparison to traditional graphical user interface widgets [4], visual code widgets provide similar functionality, but are more tangible and permanent and require different sensory-motor skills. With their unique identification number, they can automatically identify their target application, such that any input is directly sent to the target application. The user does not have to find and start the target application, but can use a generic widget interpreter application.

Visual code widgets operate on two layers of information – the device screen, which is "transparent", and the printed marker surface. Thus similar issues arise as for transparent layered user interfaces and see-through tools. Harrison et al. [6] discuss the "switching costs" for shifting attention between the two layers and the visual interference of background and foreground objects. See-through tools or toolglasses [2] are widgets that are layered between application objects and the cursor. They can be positioned over application objects in order to modify their appearance or set the context for command execution. With visual code widgets the printed widgets form the background layer and their state is superimposed over them on the foreground layer. Since there are no application objects in the view of the camera there is less visual interference. Still, textual output might interfere with the camera image in the background. Harrison et al. [7] describe a font enhancement technique, called *anti-interference font*, that mitigates this issue.

The proposed widgets are generic in the sense that they could also be implemented with other types of markers, as long as these markers provide the required orientation parameters. In particular, the code coordinate system described below is essential for detecting the focused point and for precise registration of graphical overlays. In addition, the markers and orientation parameters have to be recognized in real time in the live camera image without a perceptible delay. The markers also need to have a capacity of at least 64 bits to encode the widget type.

Other 2D marker technologies include *CyberCode* [11], *TRIP* [5], *SpotCodes* (www.highenergymagic.com), and *SemaCode* (www.semacode.org). CyberCodes and TRIP tags do not operate on mobile phone class devices. Even if they would, CyberCodes encode only 2^{24} and TRIP tags just 3^9 possible values, making it difficult to store the widget type information in the code itself. A few camera phones can read *QR Codes* [9]. These devices do not compute any orientation parameters, which are necessary for visual code widgets.

SpotCodes are a derivative of *TRIP* tags for camera phones. The system recognizes the rotation of the tag in the image, but does not seem to support the precise registration of virtual graphics with the camera image. Some interesting interaction possibilities, such as rotation controls and sliders for large displays, are described in [10]. In contrast to visual code widgets, these controls are generated on the large display and are thus not usable with non-electronic media, such as paper.

The *FieldMouse* [13] is a combination of a barcode reader and a pen-shaped mouse. The mouse detects relative movement on a flat surface. If the location of a barcode on a surface is known to the system, absolute locations can be computed by first scanning the barcode and then moving the FieldMouse. This enables simple paper-based GUIs.

3. Visual Codes

The proposed widgets rely on the visual code system described in [12]. The recognition algorithm is designed for mobile devices with limited computing capabilities and is able to simultaneously detect multiple codes in a single camera image. The codes are detectable in very low resolution images, such as 160x120 pixel view finder frames. Visual codes have a capacity of 76 bits. The data is protected by a linear code. There are C++ implementations for Symbian OS and Windows Pocket PC Phone Edition, a J2ME implementation for mobile phones, and a J2SE implementation for PCs. The J2ME implementation requires MIDP 2.0 and the J2ME camera API (JSR 135). The J2SE implementation requires the Java Media Framework (JMF) and operates with a connected camera.

3.1. Orientation Parameters

In addition to the encoded value, the recognition algorithm provides a number of orientation parameters. These include the rotation of the code in the image, the amount of tilting of the image plane relative to the code plane, and the distance between the code and the camera. No calibration step is necessary to compute these orientation parameters.

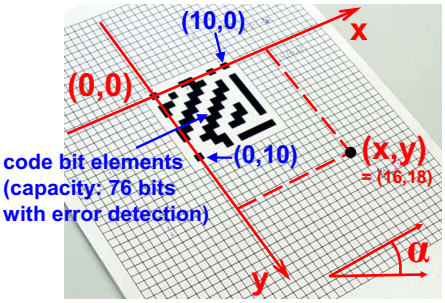

Figure 2. Each visual code defines its own local coordinate system.

An essential feature of the visual code system is the ability to map points in the image plane to corresponding points in the code plane, and vice versa. With this feature, the pixel coordinates of the camera focus, which is the point the user aims the camera at and which is indicated by a cross-hair during view finder mode, can be mapped to corresponding code coordinates. It also allows registering virtual overlays with elements in the camera image. Each code defines its own local coordinate system that has its origin at the upper left edge of the code and that is independent of the orientation of the code in the image (see Figure 2). Areas that are defined with respect to the code coordinate system are thus invariant to projective distortion.

3.2. Widget Encoding

To enable a smooth style of interaction, visual code widgets are recognized in view finder mode and updated in real time as the device is moved. In order not to delay spontaneous interaction, the type and layout of each widget is stored in the code value itself. This means that there is no initial resolution step in which the code value would have to be sent to a server, mapped to the widget description, and sent back to the device. Even with communication over Bluetooth this would incur a noticeable delay. Via the mobile phone network the delay would be much higher.

We devised a very compact way of encoding for storing the type and layout of each widget. The description of most widgets takes only 12 bits of the code, leaving 64 bits of the 76-bit value for an identifier. Application designers can freely choose this identifier and use it to establish the global application context. For use with visual code widgets, the address space is divided into four classes. The code class is stored in the two most significant bits of the code. Code class 3 is used for visual code menus, code class 2 for selection and data entry widgets, and code classes 0 and 1 are unused. The detailed layout of the encoding is given in the individual subsections.

Ideally, each widget would be universally unique. This would ensure that its semantics could be established without any other frame of reference. A person walking up to a poster with a printed widget would rely on the accompanying graphical and textual presentation to understand the semantics of the widget. A generic widget interpreter would suffice, since the unique identifier contained in the widget would then trigger a well-defined action. Unfortunately, the more complex widgets, like sliders with numeric bounds, leave less than 64 bits for the identifier. Therefore, it is difficult to encode a globally unique identifier in the remaining bits. Either more bits need to be encoded in the tag or the application context has to be established explicitly, for example, by starting a special purpose application on the phone, instead of the generic widget interpreter.

4. Visual Code Menus

Menus offer the user a list of choices which are all visible at once. Menus are imperative widgets that trigger a function upon item selection. Visual code menus encode the menu type, the number of menu items, the height of each menu item, and a 64-bit identifier. There are four types of visual code menus as shown in Figure 3: (a) single vertical menus, (b) double vertical menus, (c) circular pie menus, and (d) square pie menus. Each type has a standard layout, which means that only very few parameters need to be specified in the code. Single vertical menus, for example, are always right-aligned to a visual code and vertically centered with the code.

The height of the menu items is given in units of the code coordinate system (*ccu*). One code bit element corresponds to a single unit. The location of the individual menu items can thus be computed in terms of code coordinates directly from the code. The mapping between code coordinates and pixel coordinates allows to quickly compute the corresponding points on the screen, regardless of the perspective distortion of the code in the image. This way, the borders of the selected menu item can be superimposed over the camera image at the correct location.

The menu rendering component is generic in the sense that it just needs to interpret the menu layout that is encoded in the visual code. No information about the contents of the individual menu items is required. The menu item semantics are implicitly conveyed to the user through the

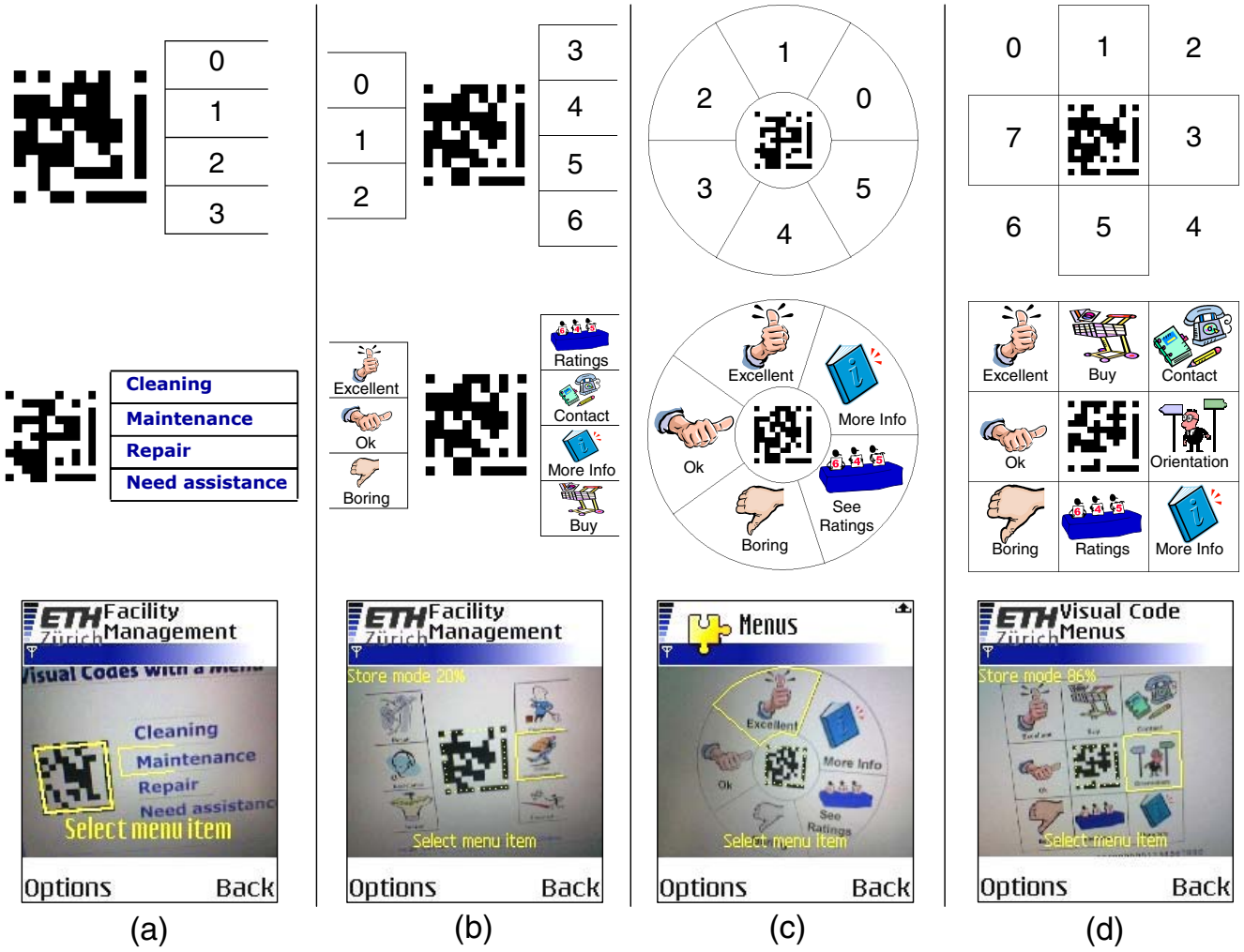

Figure 3. Visual code menus: (a) single vertical menu, (b) double vertical menu, (c) circular pie menu, and (d) square pie menu.

camera image. The lower row of Figure 3 shows example screenshots of the generic rendering component in different applications. The currently selected item is highlighted with a yellow frame. When the user completes the interaction, the resulting parameters consist of the 64-bit identifier and the index number of the selected menu item. The assignment of index numbers to menu items is shown in the upper row of Figure 3.

For devices without pen-based input, such as the Symbian phones we used, interaction with visual code menus is a two-step process. During view finder mode menu items will be selected depending on the focus point indicated by the cross-hair. Pressing the joystick button on the device will take a final high-quality picture and stop the view finder process. The camera image now freezes, the selection from the first step is retained, and the user has the opportunity to cycle through the menu selection using the joystick. One more click will submit the selected menu item.

We also implemented visual code menus on a smartphone running Pocket PC, which has a pen-based user interface. The device we used (a T-Mobile MDA III) does not provide a way to superimpose graphics over the camera image during view finder mode. We thus implemented the interaction in a slightly different way. The user first takes a picture of the menu as a whole. The picture then freezes and is shown on the display. All menu items are framed by a yellow border. The user can now select a menu item by tapping into the appropriate area.

For convenience, we have developed a Java command line tool to generate “empty” visual code menus and other types of widgets. Its input arguments are the type, number and size of the items, and the 64-bit identifier of the menu.

Its output is a PNG image containing the code and the borders of the menu items (shown in the first row of Figure 3). The idea is that an application designer uses the image in any graphics editor of her choice to label the menu items with textual or graphical content (shown in the second row of Figure 3).

4.1. Vertical Menus

In *vertical menus*, the menu items are stacked vertically upon each other. In *single vertical menus* (see column *a* in Figure 3), the items appear either to the left or to the right of the code. The table below shows the encoding of menu parameters for single vertical menus.

bits	parameter	values
75..74	code class	3 (visual code menus)
73..72	menu type	0 (single menu)
71	menu sub type	0 = left, 1 = right
70..68	item height	h = 2(x + 2) = 4..18 ccu
67..64	item count	n = x + 1 = 1..16
63..0	identifier	2^{64} values

Double vertical menus (see column *b* in Figure 3) have menu items on both sides, whose number can be specified independently.

bits	parameter	values
75..74	code class	3 (visual code menus)
73..72	menu type	1 (double menu)
71..70	item height	h = 2(x + 2) = 4..10 ccu
69..67	item count left	n_l = x + 1 = 1..8
66..64	item count right	n_r = x + 1 = 1..8
63..0	identifier	2^{64} values

4.2. Circular Pie Menus

In pie menus – also referred to as a "radial menus" – menu items are arranged as several "pie slices" around a center. As opposed to conventional linear menus, the distance to all menu items is the same and the area of an item is infinitely large, since selection only depends on the angle of the target point relative to the menu center. The location of pie menu items is easier to remember than the location of menu items in a linear list, which allows for fast access to commonly used items [3, 8]. These advantages also apply to *visual code pie* menus (shown in column *c* in Figure 3). The shape of the selected pie slice may be perspectively distorted depending on the orientation. The frame that is overlaid over the camera image is thus drawn using a polyline. The parameters are encoded as:

bits	parameter	values
75..74	code class	3 (visual code menus)
73..72	menu type	3 (circular pie menu)
71..68	outer circle radius	r = 20 + 2x = 20..50 ccu
67..64	item count	n = x + 2 = 2..17
63..0	identifier	2^{64} values

4.3. Square Pie Menus

Square pie menus (see column *d* in Figure 3) are an alternative to circular pie menus. They have up to 8 items at fixed positions around the code. As a special feature, menu items can be individually included or removed from the menu. The generated pie menu at the top of column *d*, for example, only contains items, 1, 3, 5, and 7. On the device, only the available items are selectable. A bitmap of the available menu items is stored in the code:

bits	parameter	values
75..74	code class	3 (visual code menus)
73..72	menu type	2 (square pie menu)
71..64	bitmap of used menu items	1 = used, 0 = unused
63..0	identifier	2^{64} values

5. Selection and Data Entry Widgets

Check boxes and radio buttons are selection widgets. They enable the selection of multiple options or one of a set of mutually exclusive options. Selection widgets have state and change their visual appearance depending on that state. Unlike with menus, no immediate action is associated with selection widgets. Sliders, dials, and edit fields are data entry widgets. They enable the user to provide new information to an application, rather than merely selecting information from an existing list. Here, we present visual code equipped versions of these widgets. Again, the layout of each widget is described in the code value itself.

5.1. Check Boxes and Radio Buttons

Figure 4 shows *visual code check boxes* and *radio buttons*. In a check box, multiple options can be selected, whereas radio buttons only allow for the selection of a single choice. In both cases, the list of choices is vertically stacked and appears to the right of the code. Selections are indicated by superimposing a red cross or a red bullet, respectively, over the appropriate boxes and circles in the camera image. The box or circle that is currently focused is indicated by a light yellow frame. Pressing the selection button on the handheld device selects an option, pressing it again deselects it (in the case of check boxes).

The widget rendering component on the phone remembers the state of a check box widget or radio button widget using the widget identifier that is stored in the code. When the user targets the widget again, her choice of selected items will become visible again. The state will initially be empty (for radio buttons, the first option will be selected by default) – that is, when the widget is encountered for the first time – unless its global state is retrieved via some other communication channel.

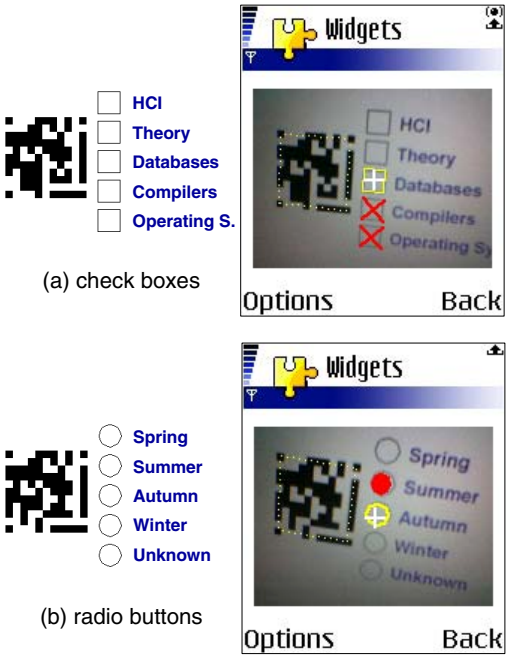

Figure 4. Visual code selection widgets: (a) check boxes and (b) radio buttons.

Interaction with check boxes and radio buttons can completely take place in view finder mode. As the device is moved, the live camera image changes and the virtual overlays are updated in real time. Pressing the selection button changes the state of the current option, but the module stays in view finder mode. Pressing the joystick button freezes the current image. On the captured widget, further selections can be performed by moving the joystick up and down. This is consistent with the behavior of visual code menus. The parameter encoding is defined as:

bits	parameter	values
75..74	code class	2 (visual code widgets)
73..72	widget type	1 (selection widget)
71	widget sub type	0 = check box
		1 = radio button
70..68	item height	h = 2(x + 2) = 4..18 ccu
67..64	item count	n = 1..16 for check boxes
		n = 2..17 for radio buttons
63..0	identifier	2^{64} values

5.2. Free-form Input

Free-form input fields have no direct counterpart in traditional graphical user interfaces. These widgets define a rectangular area next to a visual code in which the user can draw or write (with a pencil on paper). The user will then take a picture of the widget containing the drawing. The coordinate system mapping of the visual code is used for removing any perspective distortion from the captured image. The user gets a warning if the input area is not completely contained in the camera image. This is done by displaying red bars at edges which intersect the input area. As indicated with the arrow in Figure 5, the right edge intersects the input area; a red bar is thus displayed. The captured free-form input – unwarped and clipped to the bounds of the area – is then converted to a JPG image and stored on the device. It can later be submitted to a server using Bluetooth or the mobile phone network. The camera of the mobile device thus acts as a mobile scanner for selected areas of a printed document.

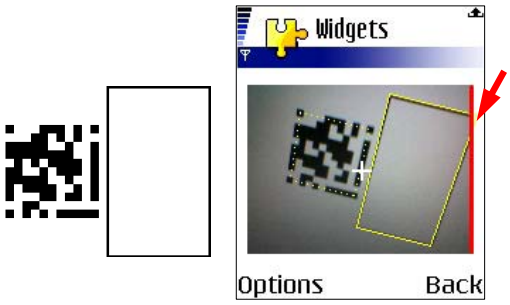

Figure 5. Free-form input widget.

5.3. Sliders

Unlike free-form input widgets, which provide "unbounded" input, *sliders* are "bounded" data entry widgets. The slider can be moved across a certain range, the selected value being proportional to the current slider position. As shown in Figure 6 there are horizontal and vertical sliders. Input can either be continuous or discrete. For a continuous slider, the sliding thumb can take any position on the scale. For a discrete slider, the thumb can only be placed at the positions of the tick marks that are indicated on the scale. There are four different options for the kind of output that is given as feedback: no textual output, percentage value (increasing or decreasing), and numeric output using the bounds and step width given in the code. Sliders are encoded as:

bits	parameter	values
75..74	code class	2 (visual code widgets)
73..72	widget type	0 (slider)
71	direction	0 = horizontal, 1 = vertical
70	mode	0 = continuous, 1 = discrete
69..68	output	0 = none, 1 = percent (inc)
		2 = percent (dec), 3 = numeric
67..62	tick distance	h = 2x + 2 = 2..128 ccu

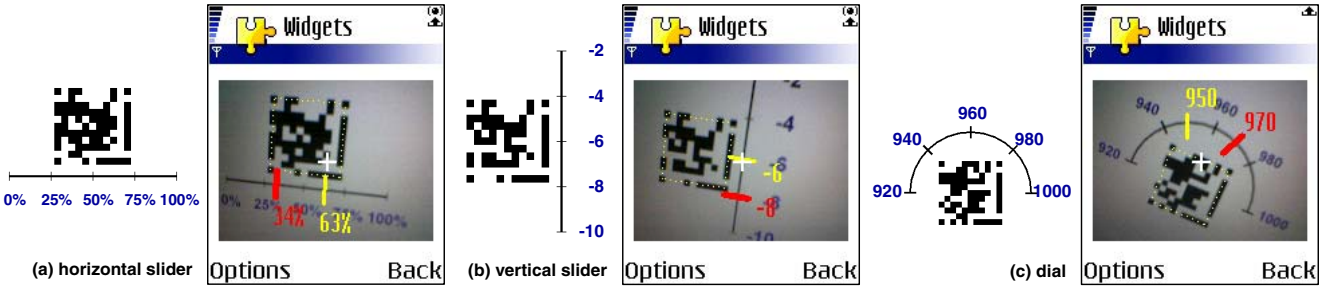

Figure 6. Visual code data entry widgets: (a) horizontal slider, (b) vertical slider, and (c) dial.

61..55 tick count n = x + 2 = 2..129

For the output modes 0, 1, and 2, the remaining bits store a 55-bit identifier. For numeric output (output mode 3), the remaining bits are structured as:

bits	parameter	values
54..39	bound	16-bit signed integer
38..23	step	16-bit signed integer
22..0	identifier	2^{23} values

Visual code sliders are most conveniently used in view finder mode. The current position is indicated with a light yellow slider thumb ("63%" and "-6" in Figure 6). The yellow thumb always immediately follows the current cursor position and moves along the scale as the camera focus changes. When the user presses the selection button, the actual value of the widget changes and the thick red slider is moved to the current slider position. The position of the red slider ("34%" and "-8") determines the value that is returned by the widget as the final result of the interaction.

5.4. Further Data Entry Widgets

Other widgets we have explored are text entry widgets, dials, and submission buttons. Clicking a *text entry widget* opens a standard GUI single-line or multi-line edit control, depending on the widget options. The widget rendering component stores the state of the widget, i.e. the already entered text. It also automatically sets the input mode to numeric or alphanumeric mode.

Visual code dials (see *c* in Figure 6) are a kind of circular slider that is controlled by the amount of rotation of the phone relative to the code. For ease of use, the scale is mapped to a semi circle above the code, instead of using the full 360°. The slider is positioned in the middle of the scale when the phone is held upright. It is positioned at the right end of the scale when the phone is rotated right by 90° and at the left end when it is rotated left by 90°. The interaction proceeds in the same way as for linear sliders.

Submission buttons belong to the imperative widget category. They are necessary to actually submit the data that was input or modified before, using the data entry and selection widgets. The widget encodes the method used for transporting the data, method-specific parameters, such as the target address, and a prefix of identifiers it is relevant for. All data that was collected from widgets with the specified prefix will be submitted to the given destination.

6. Applications

Visual code widgets can be used in a wide range of application areas. We outline two examples here: facility management and interactive television.

In the facility management and maintenance work scenario, mobile workers have a camera-equipped smartphone. At a location, machine, or other object, they find a visual code menu that contains a menu item for each step in the procedure they need to perform, such as "cleaning", "repair", or "routine maintenance". The basic facility management application we have developed just stores the selected index, the menu identifier, and a time stamp. Upon request, the application sends the stored data via SMS to a predefined phone number. Beyond this, it would be beneficial to integrate services, such as ordering spare parts, calling for assistance (based on the identifier the call could automatically be routed to an expert), and getting background information on an object. To differentiate between different phone functions – such as storing (index, identifier, and timestamp), calling, text messaging, and browsing – the identifier could be further structured: If the identifier indicates storing, the data would be stored internally; if it indicates browsing, the identifier and selected index would be used as arguments for a back-end server to retrieve the actual content. A standard set of icons for these different functions could indicate to the user what happens when a menu item is selected, showing, for example, a disk icon for storing and a phone icon for calling.

In many television programs, like in breaks of sports event broadcasts, there are quizzes in which users can select their answer from a menu of choices and send an SMS to a certain phone number. In return, they can win tickets or other prizes. Instead of requiring the user to quickly jot down a phone number that is only shortly displayed, the

camera phone can be used as an interaction device for the television program: The television screen shows a visual code menu that is captured by the user with a single click. The screen contents are then frozen on the handheld screen and the menu is available for interaction. As the television program continues, the user can take her time to cycle through the menu choices and select an answer. The result is then automatically sent as an SMS to a phone number that is encoded in the menu identifier.

The described widgets are generally useful for paper-based interaction. An example are surveys and opinion polls which could be carried out with paper forms that contain visual code widgets. The widgets are also suitable for interaction with large-scale displays for which traditional mouse and keyboard input is not feasible. This is particularly relevant for displays at public places that do not provide an input device on their own, but rely on devices that users carry with them [1].

Mobile annotation services are an active area of research. The main problem is the limited input capacity of mobile devices. It is very inconvenient to annotate an object by using the tiny keyboard on the handheld device. Most users just do not take the effort to create annotations this way. Using visual code widgets, it becomes much easier to enter selections, ratings, and free-form annotations.

7. Conclusion

Visual code widgets are a versatile mechanism to interact with elements in printed documents or on electronic screens using mobile devices. Visual code widgets could become important building blocks for marker-based interaction. For programmers, they encapsulate generic functionality and thus simplify application development. For users, they provide clear affordances and predictable behavior that is consistent across different applications.

A key feature of the proposed encoding scheme is that no initial communication is necessary. The semantics are implicitly available and conveyed to the user through the captured camera image that is augmented by virtual widget state information. A single generic widget interpreter and rendering component is sufficient to recognize all of the discussed widgets. A creation tool allows for the simple generation of different types of widgets.

As future work, we plan to use the recognition algorithm's relative camera movement detection. It does not require the presence of a visual code in the camera's field of view, but detects device movements by looking for changes in the background image. This would give more freedom in moving the device to continuously adapt parameters. We are also planning a usability evaluation of the visual code widgets with typical tasks for paper-based as well as large display interaction that more thoroughly investigates the required sensory-motor skills.

Acknowledgments

The author would like to thank Marc Langheinrich, Matthias Ringwald, and Christof Roduner for useful comments and discussions.

References

[1] R. Ballagas, M. Rohs, J. G. Sheridan, and J. Borchers. Sweep and Point & Shoot: Phonecam-based interactions for large public displays. In *CHI '05: Extended abstracts of the 2005 conference on Human factors and computing systems*. ACM Press, April 2005.

[2] E. A. Bier, M. C. Stone, K. Fishkin, W. Buxton, and T. Baudel. A taxonomy of see-through tools. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 358–364. ACM Press, 1994.

[3] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *Proceedings of ACM CHI Conference on Human Factors in Computing Systems*, pages 95–100. ACM Press, 1988.

[4] A. Cooper. *About Face: The Essentials of User Interface Design*. IDG Books Worldwide, Inc., August 1995.

[5] P. Diego López de Ipiña, R. S. Mendonça, and A. Hopper. TRIP: A low-cost vision-based location system for ubiquitous computing. *Personal and Ubiquitous Computing Journal, Springer*, 6(3):206–219, May 2002.

[6] B. L. Harrison, G. Kurtenbach, and K. J. Vicente. An experimental evaluation of transparent user interface tools and information content. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 81–90. ACM Press, 1995.

[7] B. L. Harrison and K. J. Vicente. An experimental evaluation of transparent menu usage. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 391–398. ACM Press, 1996.

[8] D. Hopkins. The design and implementation of PieMenus. *Dr. Dobb's Journal*, 16(12):16–26, 1991.

[9] International Organization for Standardization. Information Technology – Automatic Identification and Data Capture Techniques – Bar Code Symbology – QR Code. ISO/IEC 18004, 2000.

[10] A. Madhavapeddy, D. Scott, R. Sharp, and E. Upton. Using camera-phones to enhance human-computer interaction. In *Sixth International Conference on Ubiquitous Computing (Adjunct Proceedings: Demos)*, September 2004.

[11] J. Rekimoto and Y. Ayatsuka. CyberCode: Designing augmented reality environments with visual tags. In *Proceedings of DARE, Designing Augmented Reality Environments*. Springer-Verlag, 2000.

[12] M. Rohs. Real-world interaction with camera-phones. In *2nd Intl. Symposium on Ubiquitous Computing Systems (UCS 2004)*, pages 39–48, Tokyo, Japan, Nov. 2004.

[13] I. Siio, T. Masui, and K. Fukuchi. Real-world interaction using the FieldMouse. In *Proceedings of UIST'99*, pages 113–119. ACM, 1999.