Adaptive Card Design UI Implementation for an Augmented Reality Museum Application

João M.F. Rodrigues[1(✉)], João A.R. Pereira[1], João D.P. Sardo[2], Marco A.G. de Freitas[2], Pedro J.S. Cardoso[1], Miguel Gomes[3], and Paulo Bica[3]

[1] LARSyS (ISR-Lisbon) & ISE, University of the Algarve, 8005-139 Faro, Portugal
{jrodrig,pcardoso}@ualg.pt, jandrepereira00@gmail.com
[2] Institute of Engineering, University of the Algarve, 8005-139 Faro, Portugal
joao_dps@outlook.com, marcogfreitas@gmail.com
[3] SPIC - Creative Solutions, Loulé, Portugal

Abstract. Museums are great places where visitors can see, hear, touch, feel and experience interesting things. The visit is even better when visitors can select what they want to see and have ways to enhance their experience. Many museums have a huge amount of collections and objects, selecting which ones to see is sometimes difficult. A system that adapts on the fly to the user's preferences, suggesting objects that he might want to see, paths he would like to follow in their visit, as well as the complementary information he needs about each object, will be of fundamental importance. Smartphones, with their Apps are the best solution to help enhance the museum experience, nevertheless, most of the time they fail, because their user interface (UI) does not adapt to the user's preferences. This paper presents: (a) an initial framework for a museum application where augmented reality and gamification are connected with an adaptive UI, (b) an adaptive card implementation to realize the UI, and (c) an initial fast object recognition implementation for the markers used for the augmented reality.

Keywords: Apps · Adaptative UI · Marker-based AR · HCI

1 Introduction

M5SAR (Mobile Five Senses Augmented Reality System for Museums) project aims at the development of an Augmented Reality (AR) system, which consists of an application (App) platform and a device ("gadget" - hardware), to be connected to mobile devices, in order to explore the 5 human senses (sight, hearing, touch, smell and taste). The system is to be a guide in cultural, historical and museum events, complementing or replacing the traditional orientation given by tour guides, directional signs, or maps.

The number of mobile Apps, including the ones that use AR, are increasing due to the popularity of built-in cameras and global positioning systems. The massive availability of Internet connections on mobile devices also enables, the construction of personal context-aware cultural experiences [11].

M. Antona and C. Stephanidis (Eds.): UAHCI 2017, Part I, LNCS 10277, pp. 433–443, 2017.
DOI: 10.1007/978-3-319-58706-6_35

In the present and in the future, User Interfaces (UI) is a fundamental research area, where at least four (sub-)areas interconnect: Human-Computer Interaction (HCI), Artificial Intelligence (AI), User Modeling (UM) and Interaction Design (IxD). The core of the investigation in the near future should fall, most probably, in the usually called Intelligent User Interfaces (IUI) or Adaptive User Interfaces (AUI) and on the Automatic-Generation of Interfaces (AGI), connected with the best practices of IxD, user experience (UX) and Emotional UI (EUI). AUI should be enhanced with accessibility features, and can also be enhanced with AR and Gamification features.

The UIs traditionally follow a one-size-fits-all model, ignoring the needs, abilities and preferences of individual users. However, research indicated that visualization performance could be improved by adapting aspects of the visualization to the individual user [15]. As Conati et al. [8] stated, intelligent user-adaptive interfaces and/or visualizations, that can adapt on the fly to the specific needs and abilities of each individual user, is a long-term research goal. This is due to two main reasons: (a) the difficulty of extracting information about the users needs and abilities, and (b) the implementation of the UI that can adapt/change "itself" on the fly. Cortes et al. [3] define IUI as a sub-field of HCI with the goal of improving the HCI by the use of new technologies and interaction devices, including the use of AI techniques that allow adaptive or intelligent behavior. Akiki et al. [2] presented a study about adaptive model-driven UI development systems. Gajos and Weld [9] proposed an automatic system for generating UI, i.e., solution based on treating interface adaptation as an optimization problem.

Reinecke and Bernstein [12] refer that a modular UI, that allows a flexible composition from various interface elements, increases the number of variations of the interface to the power of the number of adaptable elements. Thus, instead of designing each interface from scratch, a modular user interface approach is a possible good solution, since it allows achieving many more versions with less design and implementation effort. Equally important is to adapt the UI to users with different visual, auditory, or motor impairments. Unfortunately, because of the great variety of individual incapabilities among such users, manual modular designing interfaces for each one of them is impractical and not scalable [10,13]. However, the modular and/or adaptive generation of UI offers the promise of providing personalized interfaces on the fly, but this does not mean that the user will be satisfied with his/her personalized App. According to Zhao et al. [20], the psychological process behind satisfaction is highly complex and requires a differentiation between transaction-specific satisfaction and cumulative satisfaction. Nevertheless, mobile Apps should move towards completely personalized experiences. These experiences usually are built from the aggregation of many individual pieces of content.

Having all the above in mind, at least three main challenges arise in the UI design and implementation: (a) how to harvest the necessary information about each user preferences and skills (without asking them to fill any form). (b) From the acquired information/data, how to give "intelligence" to the UI to adapt on the fly to the users changes (e.g., to the user mood). (c) How to develop this adaptive UI,

even a modular UI, without being necessary to develop a huge amount of different (sub-)modules, and at the same time still optimize the user experience (UX) and the main principles of interaction design (IxD), i.e., how to implement Automatic-Generation of Interfaces. One way these challenges can be addressed is as cards [1, 5] based UI. Card-based interaction model is not new and is now spreading pretty widely in most of the recent Apps.

This paper also focus in the implementation of AR App. The present solution is an AR marker-based method, often also called image-based [7]. AR markers-based allow adding preset signals (e.g., paintings, statues) easily detectable in the environment and use techniques of computer vision to sense them. The use of AR in museums is not new, including the implementation of head-worn displays (HWD) [17]. Other AR solutions are also available see e.g. [13]. There are many commercial AR toolkits (SDK) such as Vuforia [18] and AR content management systems, e.g. Catchoom [6], including open source SDKs, probably the most know is ARToolKit [4]. Each of the above solutions has pros and cons, some are quite expensive, others consume to much memory (it is important to stress, that our application will have many markers, at least one for each museum piece), others take too much time to load in mobile devices, etc. Here, we also focus on the initial development of an image marker detector, which will be based on the ORB binary descriptor [14].

The main contribution of this paper is a framework for the adaptive on the fly card-based UI construction, where the development of the cards has a modular architecture. In addition, an initial patch-based marker architecture for fast AR is also presented.

2 Adaptive Card Implementation

One of the objectives of this work is to develop a methodology to UIs that can adapt on the fly to each user. In particular, this section presents the architecture to create the card-based UI on run-time.

To have a full adaptive UI, we could have (at the limit) a different layout and content for each UI view and user. Nevertheless, different users could have the same layout or at least partial similarly layouts. The same layout and structure can also be used in multiple views (e.g., when showing information about different paintings to the same user), usually, in this case, the only thing that could change are the contents to display to the user. Of course, even when the layout is the same for different users the content could be different.

In this context, and with the principle of adapting the UI on the fly, makes no sense in terms of App memory and CPU optimization, to build each layout (or partial layout) from scratch every time it is required. If a layout (or partial layout) is created once, and expected to be used more than one time, this should be kept in memory instead of creating it when needed (it is important to stress that the methodology presented here was tested and developed using Unity [16] development platform).

To achieve this, we decided to separate a *view* in (A) structure/layouts and (B) contents. This means that, the application will no longer create views but

will instead make card-layouts and place different contents on the (same) card-layout at different execution points, since the (different) layouts and structures are used multiple times.

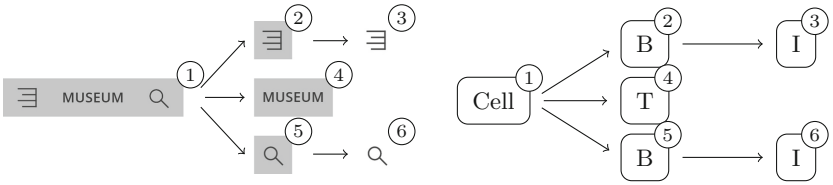
To build the structure/layout (A), an engine was created to assemble the card-layout data structure. The engine uses as input a "layout-tree" data structure, where the basic layout units, called *content format*, are joined together in *cells*, which could be joined (again) as *templates*. Both cells and templates are joined together until a card-layout is formed. Thus, each card-layout is composed by one or several cells, plus zero to several templates, that can be used in different card-layouts of the same App (the template has one or several cells, and each cell has one or several content format).

In more detail, the card-layouts are assembled in a tree structure since they represent a parent-child relationship. Figure 1a sketches the disassembled view of a card layout data structure, and the corresponding block diagram in Fig. 1b. In the figure every box represents a node, and the number in the top right corner its identifier. A tree node can be from one of three categories: (a) a content format, (b) a cell or (c) a template. Common to both the content format and the cell categories are some *properties*, like the dimensions of the node and its responsiveness behavior.

A content format (a) represents the formatting of a content (the basic unit of the card-layout), be it a text, an image, etc. Each specific content has its own properties. For example, a *text content format* (represented as T in Fig. 1b) has properties that define the font, the line spacing, the text color, etc. They also define the location where the content will appear. An *image content format* is represented by an I, and a *button content format* by a B in the same figure. A cell (b), or stack layout [19], is a node that, unlike the content formats, does not convey any information to the user, as each cell is used to organize the contents. A cell divides the children into a single line that can be oriented horizontally or vertically and gives the appropriate spacing between them. A cell child can be any of the categories aforementioned.

A template node (c), is a special node that integrates another preexisting template, i.e. a group of already structured cells and contents. This node is useful in situations where a determined structure is repeated several times, like for example the menu template shown on Fig. 1a and used in Fig. 2 (highlighted in blue). Each template can be used in any card-layouts of the App.

In the construction of the card-layout (see Fig. 2), inside the tree terminology, two terms are important: the root node (the node of the tree from which all other nodes - children - descend) and the leaf node (a node that has no children). Two rules were established and must be followed while creating a layout tree: (i) the root node ("view") must always be a cell; (ii) a leaf node must be a content or a template. Regarding (i), the root node can not be a template node, because this would mean that the new tree would be a copy of the referenced template. The root node also can not be a content format, since each template and the final card-layout should be an agglomerate of multiple contents that are organized in some shape or form. Concerning (ii), a leaf node cannot be a cell node, because



(a) Disassembled view. (b) Block diagram.

Fig. 1. Menu tree diagram.

its (cell) purpose is to arrange its children (if it has no child then it makes no sense to have it since it would be to add excessive information that needs to be sent and processed). Finally, it is important to stress the specificity of the button content format, the button itself only represents the click action and requires a child to provide visual elements to the user. These elements can be any of the categories mentioned before.

When assembling the card-layout we opted for a depth-first approach. With this approach, the card layout build engine was implemented to work in a recursive manner: (1) create the node and set its properties; (2) processes each child node (step 1) and, (3) establish the parent-child relationships (we stress again that this process was tested and developed using Unity platform [16]).

When a view, a card-layout with contents is needed, the application simply adds to the card-layout already instantiated the contents (B). If another view requires the same template it uses the same card-layout in memory and just changes the contents.

Figure 2 illustrates the build process of a card-layout. In this case, the root node of the tree is a cell that is divided vertically and whose children are represented by the dashed lines. The first child of this view is the *Menu template* as displayed in Fig. 1, but with different contents. The Menu template is assembled as follows (see Fig. 1): start by instantiating the root cell horizontally divided (node 1) and define its properties like the horizontal alignment. Next create the button content format (represented by B on node 2), followed by its child image (I on node 3). At this point the relationships are established, node 3 defines its parent as node 2, and node 2 its parent as node 1. Next, moving to the 2nd child of the Menu template root node, which in this case is a text content format (node 4), create it, specify its attributes and then set its parent as node 1. Lastly, nodes 5 and 6 are processed in a similar fashion to nodes 2 and 3. This template is now ready to be used at any time. The next node of the card-layout is a new cell node that contains an image and a text content format. The next two nodes both contain a *Field template*. This Field template is a simple template that includes two text content format arranged vertically. Finally, there is a button whose child is a cell that has a text and an image. Each of these cells follow the building principles, which were early explained.

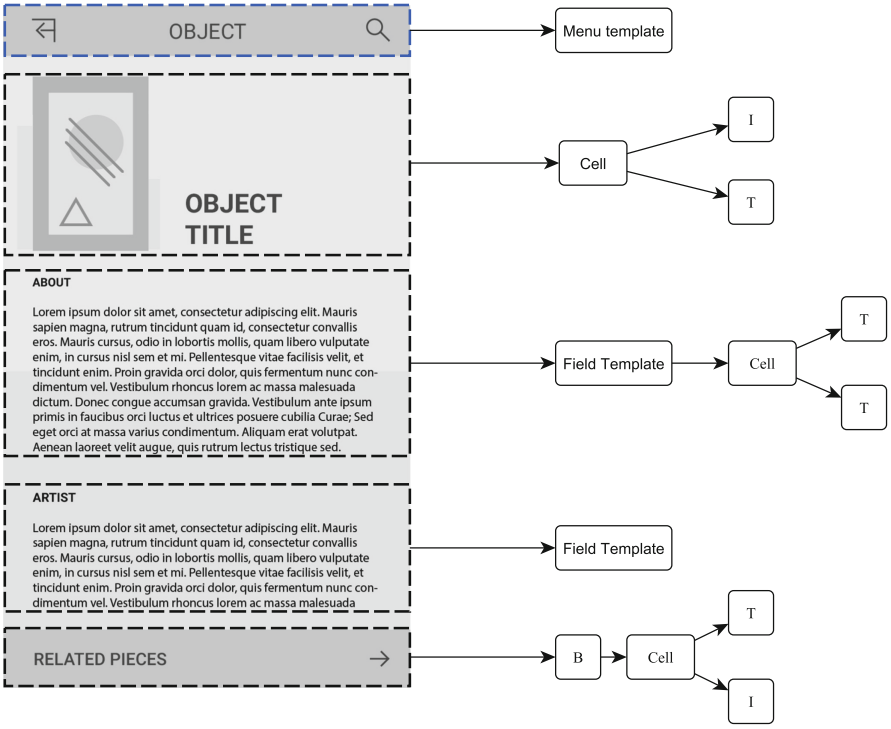


Fig. 2. Example of a museum object view.

It is very important to stress the function of the database (DB) as a fundamental component of this system, since it is where ("harvested") user information/specifications are kept, that are then converted (not presented or discussed in this paper) to the correspondent specifications for each user card-layout and card-contents (also stored in the DB). In this paper we only focus on the part of the DB related to the card-layout. The database for the card-layout implementation follows the exact same tree architecture and it can be subdivided in three major layers: (a) components, (b) formats and (c) structure.

The components layer (a) is the simplest one, where we define basic properties like colors, fonts, shadows, outlines and backgrounds. The formats layer (b) is where we indicate the type of content to be used in a child node and where we store the information related to that specific type of content, whether it is an image, a text or a button, using previously created sets of component properties. Here we can also override a particular property if needed. Then, there is the structure layer (c), where the parent-child relationships of our tree architecture are defined, node by node. It is also used to save data regarding layouts and cells, like orientation and spacing. All this information can be aggregated by templates, therefore they may be reused later, optimizing the process of creating new views.

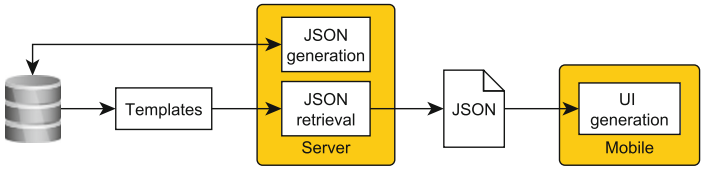


Fig. 3. UI generation overview.

When a new view has been added to the database, we need to convert it to a JSON format and store it, so that it can be requested by the application installed in the mobile device (see Fig. 3). For the JSON generation process we are running a script on the server side that receives the new template index and then connects to the database to build up the entire tree. It navigates from table to table, node by node, in the same manner that it was described for Fig. 1b. At the end of the process, the script saves the file in the DB with a time stamp, this way the App can determine whether or not that is the most recent version for that template, and if it is not, it can simply download the new JSON document. The simplified block diagram for the UI generation can be seen in Fig. 3.

3 Fast Mobile Object Detection and Tracking

In the present App a huge number of cards, the ones that describe museum objects (e.g. paintings or statues) only appear in the presence of the object, i.e., when the camera is pointed to the object. Here, we focus on object detection, recognition and tracking, with the purpose to call the respective card view.

There are many solutions (see Sect. 1) to detect a museum object and deploy the AR respective "card". Those solutions use what is called "markers" [4], which in a simplified way, are photographs (one or more) from the original object, that work as a template (see below). By using Computer Vision algorithms, they are compared with the frames captured by the mobile camera and trigger (when recognized) the identifier for the object as well as its position on the mobile screen. Here, we focus on a solution, with three goals: (a) speedup the process of downloading the markers into the mobile device, (b) do all the recognizing process in the mobile, reducing the server requirements, and (c) try to minimize power and memory consumption when doing the recognition.

Before applying our mobile object detection algorithm, museum objects were photographed and stored in a server using high quality Full HD images. Those are called image *templates* for the object. While for paintings a single photograph was used, for statues several photographs were used to represent the object. For the marker recognition implementation, it is required a reliable and fast descriptor since we aim to compute it on a mobile device. For that reason, we have opted to use the ORB descriptor [14] for object recognition.

Before we start to explain the algorithm, we define *patch* as a section of the original image, of size $N \times M$ pixels (px); see Fig. 4 top-right row. The

Fig. 4. Marker template and its patches extracted, and a marker template matched. Top to bottom, left to right: marker template (low res.), original image divided in patches, 3 most relevant patches from the full-size image, the 1/2 and from 1/4 size image. Bottom-right, object matched.

algorithm works as follows: (i) Over the template. (i.1) Compute ORB descriptor [14]; (i.2) Divide the template in patches, and extract patches with keypoints and respective descriptors; (i.3) Sort the patches by keypoint/descriptor importance, and select the K most relevant extracted patches - those are to be used as *marker patches*; (i.4) repeat steps (i.1) to (i.3), now with the image size divided by 2 and by 4 (3 scales). (i.5) Group and sort the patches from the different scales, with total number of marker patches per template, $\gamma \leq 3 \times K$ ("<=", depends on the original size of the template).

(ii) On the object recognition and tracking: (ii.1) Acquire a frame from the mobile camera and apply the ORB descriptor; (ii.2) Test the frame using the *most relevant marker patch* from the 3 scales grouping for each of the available templates (see step i.5); (ii.3) Select the template based on best classification; (ii.4) Test the object recognition using all (γ) patches from the correspondent template, matching "template-frame"; (ii.5) Object is recognized, when ratio validation threshold is verified; (ii.6) Flag if object found. After this point the object is only tracked (not tested for recognition). (ii.7) Track object based only on a valid marker patch from selected template; (ii.8) Restart the recognition process again if tracking time threshold is met.

As mentioned, in the initial step (i.1) for each object and its respective template(s), the ORB descriptor is applied for each of the 3 template scales (Full HD, 1/2 and 1/4 size). (i.2) Then, starting from the middle of the template image, each template is divided in patches (best results were obtained for $N = M = 200$ px); see Fig. 4 top-right row. The reason for starting the template division in patches from the centre, is because there is a higher probability it will have richer keypoints regions. If necessary, border regions from the template image are ignored. (i.3–4) The extracted patches are then sorted by the number of keypoints in each patch, in descending order, until $K = 5$ patches are stored per scale. This is repeated for the template with 1/2 and 1/4 of the size. This process allows farther and shorter validation distances when targeting the mobile camera onto an object. (i.5) All marker patches (γ) from the 3 scales are then grouped in descending order based on the keypoints count and stored on a object template dataset. On the mobile device, each time a frame is captured by the mobile camera (ii.1) ORB descriptor is applied, after which is matched against the object template dataset, that contains all (γ) marker patches grouped. The classifier matches the frame with the most relevant patch of each marker, (ii.2) to get the match count of similar keypoint descriptor. The marker template which has the highest match count is validated against the threshold of minimum match count ($T_{mc} = 1$) required (ii.3) for advancing to the following stage.

(ii.4) After 1 marker descriptor patch validated, all patches from the selected marker template are matched, (ii.5) and a ratio is calculated between the number of patches validated (MP_v) and the total length (γ), i.e., $r = MP_v/\gamma$. (ii.6) On ratio validation threshold validated, $T_{rv} = 0.1$, the object is recognized. After the object being recognized, and while it is in the field of view of the camera, we only need to tack it. This is a process less CPU demanding than recognition. Now, for each frame acquired, (ii.7) we only match the frame with a single valid marker patch. This steps continues until the object disappears from the field-of-view for more than $T_t = 1$ s of the camera. If this occurs then the tracking step stops (ii.8) and the recognition process starts again (steps ii.1 to ii.6).

4 Conclusions

In this paper we present an initial framework for the development of an architecture capable of producing an adaptive UI (for a museum application), the focus was on the creation process of a card-based UI, where the development of the cards has a modular architecture. In addition, it was also presented a patch-based marker architecture for mobile object recognition with application in the realm of AR.

Despite both systems being still in an initial stage of development, both present satisfactory results. For future developments, we will focus on how to harvest the necessary information about each user preferences and skills, and from the acquired information/data, how to give "intelligence" to the UI to adapt on the fly to the users changes. In the case of the mobile object recognition system, it can at the moment achieve real time recognition of 50 different objects, being the goal in the future to achieve at least 100 objects recognition in real time.

Acknowledgements. This work was supported by the Portuguese Foundation for Science and Technology (FCT), project LARSyS (UID/EEA/50009/2013), CIAC, and project M5SAR I&DT nr. 3322 financed by CRESC ALGARVE2020, PORTUGAL2020 and FEDER. We also thank Faro Municipal Museum and our project leader SPIC - Creative Solutions [www.spic.pt].

References

1. Adobe. XD Essentials: Card-based user interfaces (2016). https://goo.gl/gg8qUM. Accessed 16 Nov 2016
2. Akiki, P.A., Bandara, A.K., Yu, Y.: Adaptive model-driven user interface development systems. ACM Comput. Surv. **47**(1), 9:1–9:33 (2015)
3. Alvarez-Cortes, V., Zayas, B.E., Uresti, J.A.R., Zarate, V.H.: Current Challenges and Applications for Adaptive User Interfaces. INTECH Open Access Publisher, Rijeka (2009)
4. Artoolkit. Artoolkit, the world's most widely used tracking library for augmented reality (2016). http://artoolkit.org/. Accessed 16 Nov 2016
5. Babich, N.: Designing card-based user interfaces, smashing magazine (2016). https://goo.gl/AM46gT. Accessed 18 Nov 2016
6. Catchoom. Catchoom (2016). http://catchoom.com/. Accessed 16 Nov 2016
7. Cheng, K.-H., Tsai, C.-C.: Affordances of augmented reality in science learning: suggestions for future research. J. Sci. Educ. Technol. **22**(4), 449–462 (2013)
8. Conati, C., Carenini, G., Toker, D., Lallé, S:. Towards user-adaptive information visualization. In: AAAI, pp. 4100–4106 (2015)
9. Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: Proceedings of International Conference on Intelligent User Interfaces, pp. 93–100. ACM (2004)
10. Gajos, K.Z., Wobbrock, J.O., Weld, D.S.: Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In: Proceedings of SIGCHI Conference on Human Factors in Computing Systems, pp. 1257–1266. ACM (2008)
11. Jung, T., Chung, N., Leue, M.C.: The determinants of recommendations to use augmented reality technologies: the case of a Korean theme park. Tourism Manag. **49**, 75–86 (2015)
12. Reinecke, K., Bernstein, A.: Knowing what a user likes: a design science approach to interfaces that automatically adapt to culture. MIS Q. **37**(2), 427–453 (2013)
13. Rodrigues, J.M.F., Lessa, J., Gregrio, M., Ramos, C., Cardoso, P.J.S.: An initial framework for a museum application for senior citizens. In: Proceedings of 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (2016)
14. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: an efficient alternative to SIFT or SURF. In: Proceedings of International Conference on Computer Vision, pp. 2564–2571. IEEE (2011)
15. Steichen, B., Conati, C., Carenini, G.: Inferring visualization task properties, user performance, and user cognitive abilities from eye gaze data. ACM Trans. Interact. Intell. Syst. **4**(2), 11 (2014)
16. Unity. Unity 3D (2014). https://unity3d.com/pt. Accessed 10 Nov 2014
17. Vainstein, N., Kuflik, T., Lanir, J.: Towards using mobile, head-worn displays in cultural heritage: user requirements and a research agenda. In: Proceedings of 21st International Conference on Intelligent User Interfaces, pp. 327–331. ACM (2016)

18. Vuforia. Vuforia (2016). https://www.vuforia.com/. Accessed 16 Nov 2016
19. Xamarin. Stack layout - Xamarin (2016). https://goo.gl/i7LhG9. Accessed 18 Nov 2016
20. Zhao, L., Yaobin, L., Zhang, L., Chau, P.Y.K.: Assessing the effects of service quality and justice on customer satisfaction and the continuance intention of mobile value-added services: an empirical test of a multidimensional model. Decis. Support Syst. **52**(3), 645–656 (2012)