ARCHIE: A User-Focused Framework for Testing Augmented Reality Applications in the Wild

Category: Systems

ABSTRACT

In this paper, we present ARCHIE, a framework for testing augmented reality applications in the wild. ARCHIE collects user feedback and system state data in situ to help developers identify and debug issues important to testers. It also supports testing of multiple application versions (called "profiles") in a single evaluation session, prioritizing those versions which the tester finds more appealing. To evaluate ARCHIE, we implemented four distinct test case applications and used these applications to examine the performance overhead and context switching cost of incorporating our framework into a pre-existing code base. With these, we demonstrate that ARCHIE provides no significant overhead for AR applications, and introduces at most 2% processing overhead when switching among large groups of testable profiles.

Index Terms: Software and its engineering—Software testing and debugging; Human-centered computing—User interface toolkits; Human-centered computing—Mixed / augmented reality

1 INTRODUCTION

Augmented reality (AR) applications are increasing in popularity, with applications moving beyond just gaming to fields such as healthcare [47, 61, 72], education [39, 51, 64, 67], manufacturing [32, 54, 63, 67], and so on. On mobile application marketplaces, AR apps for entertainment [12, 24, 27], measurement [7, 13], translation [11, 15], and more are reaching millions and hundreds of millions of downloads. While augmented reality systems have been around since the early 1990s [68], the current rise in popularity is due to two reasons. The first reason is the hardware able to support AR is relatively inexpensive and widely available. The market niche originally dominated by bulky and expensive head-mounted displays (HMDs) such as Google Glass [8], Microsoft Hololens [15], and Oculus Rift [19] is now being filled by conventional smartphones, particularly when paired with inexpensive headsets such as Google Cardboard [9]. Even older devices such as the Google Nexus 5 [12], originally released in 2013, are able to download an AR app as easily as a regular smartphone app.

The second reason is that AR-specific development kits (such as Google’s ARCore for Android [10] and Apple’s ARKit for iOS [2]) and integrated development environments (such as Unity3D [32] with Vuforia [36]) have made it easier for developers to build AR apps without specialized knowledge of machine learning, computer vision, or graphics. Developers who desire greater control over their systems can make use of computer vision libraries such as TensorFlow [29] or OpenCV [21] to create even more powerful applications.

However, the current AR software development landscape is lacking any software tools to test and debug AR apps. AR apps are different from conventional apps in that the augmented images and labels are generated and positioned based on the user’s behavior and environment. For example, a navigation app like Google Maps AR [37] will need to determine the user’s context from their position, facing direction, and visible entities in their environment to determine what directions to place on the screen and where. This makes AR applications very difficult to test in the wild due to high variability of user behavior and environmental factors. User movement speed, weather, lighting, presence and movement of bystanders, and many other factors all contribute to the performance of an AR system. However, traditional software engineering practices such as the creation and automation of unit tests are ill-equipped to capture and recreate these conditions, as the test writers would be unable to predict which combinations and interactions of conditions a user would encounter. Similarly, because these conditions are likely to occur in tandem, logic to handle one set of conditions is likely to interfere with the logic for another set, making regression testing difficult as well. Once the evaluations are over, even with detailed interviews or questionnaires filled out by the testers, it is impossible to know what underlying errors in the system caused the observable issue. For example, incorrect label placement in an AR navigation app could be due to a lag in the output generation or an error in the placement logic.

To help address these needs, we present ARCHIE, the Augmented Reality Computer-Human Interaction Evaluator framework, specifically designed to help AR application developers with system testing and the collection of feedback and debugging data. Our framework makes the following contributions:

• Identifying performance and usability issues in the wild. ARCHIE collects performance and usability feedback from testers using a smartwatch-based companion app. Feedback is gathered after specific system events to verify positive results (such as identifying an AR target), and after periods of inactivity to verify negative results (such as failing to identify a target). Testers are also able to identify any specific issues they observe (such as slow output, poor output placement, etc.)

• Recreating and debugging run-time errors. When testers submit poor performance or usability feedback, it is important to understand what underlying errors caused that poor feedback. ARCHIE aggregates environmental and system state data during run-time, and provides collections of that data for each instance of poor feedback submitted by the tester. Developers can then query that data, or feed it to the app back in the lab to identify and debug system errors.

• Testing multiple implementations. ARCHIE also allows developers to implement multiple logical modules, called profiles, which can be used to compare different functional options for an application, such as user interfaces or event recognition. ARCHIE instantiates these profiles at run-time, and interleaves them for the tester, collecting dedicated feedback for each one.

The rest of the paper is organized as follows: Section 2 provides an overview of current issues facing AR application testers and the evaluation methods they employ. Section 3 describes the design of ARCHIE, and Section 4 describes how developers and testers interact with ARCHIE. Sections 5 and 6 evaluates our solution, and discusses how to extend ARCHIE to support virtual reality (VR) app testing respectively. Section 7 discusses the related work, and Section 8 concludes.

2 OBSERVATIONS OF REAL-WORLD TESTING PRACTICES

An AR app has some common errors that are difficult to detect using conventional software engineering testing. To illustrate, consider an AR app for smartphones that seeks to warn distracted pedestrians...
Online Submission ID: 0

(a) Classifier error generates wrong warning text
(b) Slow processing, quick user movement prevents classification
(c) Poor label placement covers pertinent real-world content (car in intersection)
(d) Alert correctly placed but difficult to read due to poor color choice

Figure 1: Conditions that may negatively impact users’ experience of “distracted pedestrian” AR application.

The first kind of error are **classification errors**, as seen in Figure 1a. These are errors that result in incorrect label or image to be displayed. The causes of classification error can be due to the mismatch between the training and testing data sets used in the lab, and the actual visual inputs encountered in the wild. Some contributors to classification errors include environmental conditions such as poor light and viewing angle, as well as intra-class variations (such as different breeds of the same animal or different models of the same car). Because AR systems are subject to user movement, classification errors can also occur when the tester is moving too quickly to get a clear image of the target (such as Figure 1b). As it is impossible to train a classifier for every variation of inputs that it may receive, or to control exactly the movement and behavior of the user, classification errors are very common for AR systems.

The second kind of error are **placement errors**. Placement errors occur when the system has correctly classified the visual input, but places the label or image output in an inappropriate location on the display. This kind of error has the potential to be dangerous to testers, as they can affect the tester’s safety in the real world. For example, Figure 1c shows how the stop sign can be correctly identified, but the warning label blocks the tester’s view of the vehicle in the intersection.

The third kind of error are **resource limitation errors**. These are errors that are caused by insufficient resources (e.g. CPU, RAM, etc.) in the underlying system. It is possible for resource limitation errors to contribute to a classification or placement error. For example, the classification error shown in Figure 1b may be due to a slow sampling rate from the device camera, or insufficient resources to process the input in time.

The fourth kind of error are **style errors**. Style errors occur when the system classifies input and generates output correctly, and places the output in an appropriate location, but presents the output in an inappropriate way. For example, in Figure 1d the warning label for the stop sign is placed above the intersection, but the white text is unreadable against the bright sky.

![Figure 1](image1.png)

Figure 1: Length of evaluation

![Figure 2](image2.png)

Figure 2: Evaluation trends of surveyed AR systems

<table>
<thead>
<tr>
<th>Environment</th>
<th>System Function</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab</td>
<td>Label display</td>
<td>71, 81, 82</td>
</tr>
<tr>
<td></td>
<td>Label placement</td>
<td>56, 65, 77</td>
</tr>
<tr>
<td></td>
<td>UI comparison</td>
<td>45, 66</td>
</tr>
<tr>
<td></td>
<td>Interaction</td>
<td>40, 44, 49, 62, 76</td>
</tr>
<tr>
<td></td>
<td>Avatars</td>
<td>53, 80, 86</td>
</tr>
<tr>
<td>Wild</td>
<td>Label display</td>
<td>41, 60, 79</td>
</tr>
<tr>
<td></td>
<td>Interaction</td>
<td>46, 88</td>
</tr>
</tbody>
</table>

Table 1: System functions targeted by surveyed AR systems

2.1 Survey of Feedback Methods

To understand how AR researchers handle testing errors, we conducted a survey of recent AR related papers. We first identified the high level system function being tested. We then categorized papers by whether the project was tested in the lab or in the wild, the length of the evaluation, the size of the test group, and primary method of collecting tester feedback. The results are presented in Table 1 and Figure 2 and summarized into the following key points.

- **Head-mounted displays (HMDs) are a popular testing platform.** The majority of papers surveyed (62%) tested their platform either solely on HMDs or supplementing the HMD with a smartphone. A comparatively small number (24%) examined HMDs solely on smartphones, while 14% tested on a custom platform.

- **Questionnaires and interviews are the primary methods for gathering feedback from testers.** Of the papers surveyed, 81% relied solely on questionnaires, interviews, or both to collect tester feedback. The remaining papers tested on a custom platform.


Most tests are conducted in a single sitting with a small group of testers. The bulk of papers surveyed (62%) conducted all tester activities within a single day. Of those papers, 69% conducted their evaluations with 25 or fewer testers.

A non-trivial portion of papers explicitly asked testers to compare user interface (UI) options. More than one third of papers surveyed [45, 55, 56, 66, 77, 78, 85] gave testers multiple UI options to evaluate, and took feedback specifically on which UI version was preferred for the target scenario.

2.2 Limitations of Existing Methods

From this survey, it is clear that questionnaires, interviews, or a combination of both are the method of choice for eliciting feedback from system testers. Unfortunately, these methods suffer from a number of limitations. The first limitation is that, while questionnaires and interviews can be very helpful in identifying whether the issues discussed above (classifier, placement, resource limitation, style) occurred, they are limited in their ability to identify the issues' underlying cause. For example, a tester might indicate in her questionnaire that the application failed to identify the stop sign, but would not necessarily be able to tell you why. Was the tester moving too quickly for the camera to get a clear picture, as shown in Figure 2? Was she viewing the target from too severe of an angle for the classifier to recognize it? Was the light too low? Was the classifier too slow to process the image, and was overridden by earlier or later inputs? Using questionnaires and interviews alone cannot help researchers answer these questions.

The second limitation is that questionnaires and interviews provide no support for recreating error scenarios upon returning to the lab. Subjective, written responses from testers provide no actionable data to use as input back into the system. Any data sets, system inputs, or generated outputs must be recreated from scratch in order to debug the issues reported by testers.

The third limitation is that, when applications are testing multiple GUIs or implementation versions, questionnaires and interviews cannot guarantee that testing conditions for each version are consistent. Specifically, if a tester claims to prefer Interface A over Interface B, it is important to verify that the tester’s preference is due to the differences in interfaces only, and not because of additional factors. For example, if the tester only evaluated Interface A during the day and Interface B at night, it is possible that the preference was impacted by time of day. Utilizing questionnaires and interviews alone cannot help researchers answer these questions.

Based on our observations of current testing methods for AR systems, and the inability of these methods to handle the testing issues discussed above, we designed ARCHIE to (1) collect tester feedback in situ to assess system performance and usability, (2) collect system input and resource consumption data to assist developer debugging efforts, and (3) facilitate multi-UI testing to support a more consistent tester experience.

ARCHIE’s lightweight design allows it to provide these features without impacting the function of the application under test. The following sections discuss the design of ARCHIE in greater detail.

3 System Design

3.1 Framework Architecture

As shown in Figure 3, the ARCHIE framework consists of three primary components: the ARCHIE application, as developed by the external research team, the ARCHIE annotator helper application, and the ARCHIE core library. The annotator helper application is a lightweight client responsible only for forwarding input requests to the evaluator, and returning his or her response to the main system. The core library contains the bulk of our framework functionality. Any application wishing to utilize the ARCHIE framework may also implement modules called profiles to represent a distinct UI design point for testers to evaluate. A profile could contain logic for input, output, or core event recognition (such as gesture recognition or object classification).

Figure 3: ARCHIE framework system architecture

An AR application-under-test (AUT) interacts with the core library by passing a configuration file on start-up to the ARCHIE Controller, the primary interface point between testable applications and the rest of the framework. The configuration file should minimally contain the list of input, output, and core event recognition profiles the framework is expected to utilize during the test instance. The ARCHIE Controller then instantiates these profiles, raising the appropriate events to connect to input sources, retrieve and process input data, and communicate the results to the user. Once the configuration file settings have been imported and the profile event handlers are initialized, the controller is responsible for managing the profile switching schedule, interfacing with the annotator application, and monitoring the appropriate device-level resources over the remainder of the test instance. When the evaluation is complete, ARCHIE generates user feedback and system state output logs for researchers to review.

3.2 Profile Scheduling

The time spent conducting trials is a precious resource and should be used as efficiently as possible. The use of profiles allows researchers to test multiple types of user interactions over a given trial with the same test group. The straightforward solution for choosing which profile to execute next would be to simply execute each profile in Round Robin order. However, this is not an effective use of a tester’s time, as it could obligate the tester to spend a large portion of the trial interfacing with system features that may not be applicable or interesting to her. Instead, ARCHIE provides a customized scheduling...
Algorithm 1 Profile Scheduler

```
1: P, L\_P, T, U, C ← configuration file
2: if t == 0 then
3:     for \( \rho \) in P do
4:         \( Q_t(\rho) = IR_t = \alpha(\rho_t) \)
5: else
6:     while size(P) > L\_P do
7:         for \( \rho \) in P do
8:             \( IR_t = \alpha(\rho_t) \)
9:         \( Q_t(\rho) = Q_{t-1}(\rho) + (\frac{T}{t})(IR_t - Q_{t-1}(\rho)) \)
10:        \( \rho_{\min} ← \min(Q_t(P)) \)
11:        \( \rho_{\max} ← \max(Q_t(P)) \)
12:        if diff(\( \rho_{\max}, \rho_{\min} \)) ≥ C then
13:            remove(P, \( \rho_{\min} \))
14:        scheduleNextProfile(\( \rho_{\max} \))
15:     while executionTimeRemaining do
16:         roundRobin(P)
```

To implement this algorithm, we utilized a customized solution for the Multi-Arm Bandit Problem as shown in Algorithm 1. We first define the reward function for a given data vector collected from profile \( \rho \) at instance \( i \) within time slice \( t \) as shown in Equation 1.

\[
A(\rho_i) = \begin{cases} 
1 + usrScore_i & \text{sysResp}_i = usrResp_i \\
usrScore_i & \text{sysResp}_i \neq usrResp_i 
\end{cases} 
\]  

(1)

\[
\alpha(\rho_i) = \sum_{i=0}^{T} A(\rho_i) 
\]  

(2)

Once all the instantaneous rewards have been calculated, the overall reward earned by the profile during the given time slice is calculated using Equation 2. It should be noted that both of these equations are simply default functions provided by ARCHIE and that researchers wishing to use our framework may provide their own functions that are better tailored to their specific needs.

We then customize the traditional upper-confidence-bound solution of the Multi-Arm Bandit Problem, setting values for the parameters in Table 2 using entries retrieved from the application’s configuration file. The algorithm starts with the entire collection of profiles \( P \) as listed in the configuration file and calculates the action-value function \( Q_t(\rho) \) for each profile \( \rho \) at the end of a given time slice \( t \). Since there is no guarantee that a user will always give the same feedback for a given profile, the action-value function is calculated in a non-stationary manner, giving greater weight to the most recent reward result for that profile. If the difference between the minimum and maximum action-values for a given time slice rises above a developer-defined threshold \( C \), the profile that earned the minimum score will be eliminated from the collection.

Once the count of profiles remaining in the collection reaches a developer-defined limit \( L\_P \), ARCHIE will execute the remaining

Table 2: Parameter list for profile scheduler algorithm. Values can be set in the configuration file for the application-under-test.

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>Collection</td>
<td>The set of profiles, classifiers to be evaluated</td>
</tr>
<tr>
<td>( L_P )</td>
<td>Integer</td>
<td>( P ) min-size constant</td>
</tr>
<tr>
<td>( T )</td>
<td>Integer</td>
<td>Time slice size constant</td>
</tr>
<tr>
<td>( U )</td>
<td>Float</td>
<td>Uncertainty constant</td>
</tr>
<tr>
<td>( C )</td>
<td>Float</td>
<td>Confidence threshold constant</td>
</tr>
<tr>
<td>( t )</td>
<td>Object</td>
<td>Current time slice of size ( T )</td>
</tr>
<tr>
<td>( \rho_t )</td>
<td>Collection</td>
<td>Set of user response data collected while executing profile ( \rho ) over time slice ( t )</td>
</tr>
<tr>
<td>( IR_t )</td>
<td>Integer</td>
<td>Reward received at time ( t )</td>
</tr>
<tr>
<td>( Q_t(\rho) )</td>
<td>Float</td>
<td>Action-value function for profile ( \rho ) at time ( t )</td>
</tr>
</tbody>
</table>
profiles in Round Robin fashion for the remainder of the trial.

The goal of this customized algorithm is to prioritize those profiles in which the tester shows the greatest interest, while simultaneously suppressing those profiles which the tester finds inferior. If a profile’s reward is low enough that it falls below the acceptability threshold, then it is removed entirely and the tester does not have to interact with it any more. Furthermore, the algorithm does not require the system to converge on a single best answer. By providing a profile collection limit, the developer is able to designate the point at which the system will stop culling the collection and continue with Round Robin instead. By combining developer-provided parameters with user-driven reward functions, this scheduling algorithm helps ensure that testers are only presented with system options in which they are interested, and none in which they are not.

<table>
<thead>
<tr>
<th>Data Group</th>
<th>Fields</th>
<th>Collect</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Recognition: Input</td>
<td>Timestamp; Preprocessed feature sets</td>
<td>1A, 2A, 2B</td>
<td></td>
</tr>
<tr>
<td>Event Recognition: Results</td>
<td>Timestamp; Result; Preprocessing time; Processing time</td>
<td>1B, 2A, 2B</td>
<td></td>
</tr>
<tr>
<td>System Stats: Display</td>
<td>Timestamp; Frame count; Total frame time; Average frame time</td>
<td>3A, 3A, 5</td>
<td></td>
</tr>
<tr>
<td>System Stats: Resources</td>
<td>Timestamp; CPU consumption; RAM consumption; Process info</td>
<td>3B, 3B, 5</td>
<td></td>
</tr>
<tr>
<td>UX Feedback</td>
<td>Timestamp; UX feature; User response</td>
<td>4A, 4B</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Data collected by ARCHIE framework at various "collect" and "commit" points shown in Figure 4.

3.3 System State Data Collection

Based on the lessons learned in Section 2, we know how important it is to provide some way of identifying and debugging unintended event recognition behavior while conducting in-the-wild tests. ARCHIE helps researchers by buffering and storing collections of system state and user response data, clustered according to events in which researchers are interested, such as event recognition results which do not agree with the user’s ground truth labels, or poor usability feedback.

4 Testing with ARCHIE

4.1 Collecting Tester Feedback

Once the system has been initialized by the developer with the profiles that should be evaluated, the system can be handed off to testers. The ARCHIE framework requests input from the tester (1) to confirm when an event of interest is identified, (2) on a developer-configured schedule to see if any events of interest have been missed, and (3) on a second developer-configured schedule to collect general usability feedback. An input request starts by playing an alert on the annotator device (such as a smartwatch, as shown in Figure 5), and presenting the question set by the developer (such as “Have you seen a stop sign in the last 10 minutes?” or “How are you liking the app so far?”). The user can toggle between positive and negative responses by tapping the center of the screen (Figure 5a). If the developer has included a request for additional input (such as rating how much the app is liked or disliked), the tester can drag her finger around the edge of the screen to provide her input (Figure 5b). Finally, the tester can select any applicable issues she encountered by checking them off from a grid of options, as shown in Figure 5c.

4.2 Debugging with Output Files

Once a tester has provided system and usability feedback, the output files generated by the ARCHIE framework can be used to recreate and debug the original scenario in order to determine the underlying cause. Consider again Figure 1 in which a tester is evaluating a smartphone-based application to warn distracted pedestrians of imminent intersections. The tester has completed the evaluation, and reported back to the research team that the system was a little slow in providing warnings of stop signs, and sometimes missed the stop signs altogether. At this point, the researchers have no way of knowing what caused this behavior. To find out, they retrieve the...
data logs written by ARCHIE to the device file system. The data files (as described in Table 3) are organized into timestamped directories according to the system event that prompted a user response, such as an asynchronous event query (“async”), a system-generated event recognition verification (“system”), or a user experience feedback request (“ux”). The directory name is then appended with the user’s response (such as “falseNeg”, “falsePos”, or “slow”), and the date and time the response was provided. Inside each directory are the relevant files associated with that event, such as the inputs provided to the system and results that the system generated, as well as system display and resource consumption logs.

From these log files, it is possible to determine what unique combination of internal and external conditions caused the system to generate false positive or negative classifications, or contributed to slow performance. Input frames can be fed back into the system in the lab to determine where a classification error occurred or where an inappropriate label was placed. Reviewing the system log can help identify whether a spike in RAM and CPU consumption was anomalous or consistent with the rest of the trial. Exploring the results and system display logs in tandem can help determine whether the time an augmentation takes to display is due to the application’s processing time or lag in the device as a whole.

5 Evaluations

We implemented and evaluated ARCHIE using a Google Pixel 2 running Android 8.1.0. We elected to test our framework using a smartphone because it could also double as an HMD when placed in an appropriate head mount. For testing purposes, we are primarily interested in the performance impacts when incorporating ARCHIE into different types of AR applications, to ensure that any negative feedback a tester may provide is due to the performance of the application itself, and not the influence of ARCHIE.

The design for the ARCHIE framework supports the collection of camera frames, video clips, audio clips, GPS and IMU data as sources of input for the data logs. Currently, only support for camera frame collection has been implemented because it was sufficient to meet the needs of our evaluations. The framework design also supports the collection of output camera frames containing the final augmented view displayed the user, but was not implemented due to its library-specific (ARKit, Vuforia, etc.) nature. We anticipate adding this functionality to the framework in the future when more implementation-agnostic API support becomes available.

5.1 Evaluation Candidates

We used the following four case study applications to evaluate ARCHIE: TF Classify, TF Speech, Color Blob Detector, and DroneTracker. These applications were chosen to showcase how our framework integrates with three of the most popular computer vision support libraries in research and industry: TensorFlow [29], OpenCV [21], and Vuforia [36]. Each application was either taken from the publicly available sample applications published by the library developers or created as part of a tutorial, and represents a range of sensor and UI interactions.

“TF Classify” [30] (shortened to “TF-C”, shown in Figure 6a) is an image classification application distributed and maintained by the TensorFlow development team which utilizes preview frames from the device’s camera, categorizes the primary subject of the image, and populates a text box on the phone screen with the results in real-time. The camera preview is not manipulated directly; only the content of the text box is updated.

“TF Speech” [30] (shortened to “TF-S”, shown in Figure 6b) is another application from TensorFlow which focuses on audio processing rather than image processing. Instead of using the camera, it captures audio snippets from the microphone and parses them in order to identify words it recognizes. The list of known words is...
5.2 Overhead of Incorporating ARCHIE

Because each of the evaluation candidate applications differs in nature from its fellows, each one had to be tested differently. TF-C was placed on the edge of a desk, pointed downward at a rug with a design detailed enough to be confused between two of the classifier’s known classes. TF-S was laid on the desk next to an active radio. CBD was placed on the edge of the desk and pointed toward the rug as well, with a single screen touch by the tester to activate the color blob boundary recognition logic. DT was placed on the edge of a stack of books, overlooking the two targets. Each application was then allowed to run for five trials of five minutes each, while ARCHIE collected the appropriate system state and application output data.

Figures 7 and 8 reflect the impact to UI and system performance respectively when incorporating the ARCHIE framework into an existing application. While examining impacts to CPU and RAM consumption is expected, we also included the UI metrics of display FPS and time to classify in order to demonstrate how ARCHIE might impact the user experience. If refactoring an application resulted in lowered FPS or a dramatically increased classification time, then developers could expect users to experience a decreased quality of overall system performance due to delayed or inaccurate system responses. However, no ARCHIE case studies showed significant impacts to FPS or classification time.

Of the performance metrics evaluated, the only case in which performance showed a significant hit was when ARCHIE was added to the TF-Speech app. In this case, user-space CPU utilization went up from about 28% to about 40%. We believe that the reason for this is due to the extremely simple nature of the original app meaning that adding ARCHIE introduced a non-trivial amount of additional processing. The other apps utilize the camera and some degree of image processing (operations which already require a good deal of resource consumption), such that introducing the additional processing for ARCHIE was negligible.

5.3 Cost of Switching Profiles

Providing context-switching ability for a collection of profiles and classifiers is a key contribution of the ARCHIE framework. However, this feature provides no benefit if the cost of context-switching is high. Therefore, we ran a series of tests evaluating the CPU and RAM consumption overhead of context-switching for collections of profiles ranging in size from one to ten, as reflected in Figure 9. TF-C was used for this evaluation. Five configurations were created with various numbers of profiles (one, three, five, seven and ten) for ARCHIE to manage, with each profile representing a distinct instance of the core image recognition classifier. Each profile was displayed on the application screen, and when a recognized word is heard, the application will briefly highlight the corresponding box for that word on the screen. 

“Color Blob Detector” (shortened to “CBD”, shown in Figure 6c) is an application distributed by the OpenCV development team which performs active color processing and boundary detection on a live camera preview. To detect color blobs, the user must press the device screen to select an area of the camera preview that he or she wishes to match. The application then segments that portion of the preview frame and identifies the primary color. For all subsequent preview frames until the user selects a new color blob, boundaries of areas within the preview that match the same primary color are detected and highlighted in real-time.

“DroneTracker” (shortened to “DT”, shown in Figure 6d) is an application that we obtained from [13], as there are no formally recognized sample applications available for Vuforia. The Drone-Tracker app consists of a dual-target tracking system which displays a small sci-fi drone that responds to and follows a target moved by the user around the screen in real-time.

“Tracker” app consists of a dual-target tracking system which displays application output data.
allowed to execute for one minute before switching to the next; each configuration was executed five times for 25 minutes per execution.

During testing, however, the camera interface would occasionally freeze, causing a significant drop in resource consumption until the next context switch, at which point the camera preview would refresh and processing began again. Over all of the trials for all configurations, only 5% of the records were affected and removed. The results in Figure 9 reflect the aggregated results for the remaining 95%. At most, resource consumption fluctuates by 2% when considered across five trials of 25 minutes each for each of the five test cases. This leads us to conclude that ARCHIE provides no significant overhead when switching between profiles during evaluations.

6 ADDITIONAL DISCUSSION

Even though ARCHIE was originally designed with AR applications in mind, it could easily be adapted to work with virtual reality (VR) applications. We expanded our survey from Section 2 to examine VR systems as well (results in Table 1), and observed the following: (1) all evaluations were conducted in the lab on some combination of smartphones and HMDs, (2) all papers but one 38 used questionnaires and interviews to collect user feedback, and (3) three papers 32, 78, 83 conducted their own explicit UI comparisons. Based on this, we believe ARCHIE has real potential for use in testing VR applications as they move out of the lab and into the wild. To accommodate this shift, we would need support for new profiles to monitor motion sickness, and data collection from new sensors such as galvanic skin response.

![Table 4: Expanded survey of contemporary VR systems](image)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>Questionnaire + Interview</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>Questionnaire</td>
<td>375</td>
</tr>
<tr>
<td>UI comparison</td>
<td>Questionnaire + Interview</td>
<td>178</td>
</tr>
<tr>
<td>Interaction</td>
<td>Questionnaire + Interview</td>
<td>159, 83</td>
</tr>
<tr>
<td></td>
<td>Questionnaire</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>Interview</td>
<td>179</td>
</tr>
<tr>
<td>Avar</td>
<td>Questionnaire</td>
<td>157</td>
</tr>
<tr>
<td>Smart phone</td>
<td>Interaction</td>
<td>Manual coding</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>Interview</td>
</tr>
<tr>
<td></td>
<td>Interaction</td>
<td>Questionnaire</td>
</tr>
</tbody>
</table>

7 RELATED WORK

Many tools and frameworks exist to help developers test their mobile AR systems. These testing platforms serve a number of goals, many of which can be supplemented with ARCHIE.

Commercial testing tools. The increase in commercially available AR hardware and development environments brings with it an increase in commercial testing tools as well. Oculus provides official guidelines for performance optimization 35 as well as an array of performance monitoring and debugging tools 6,18,23,25 for applications built for the Rift headset. Microsoft provides its own official guidelines for testing applications built for the HoloLens headset 31, in addition to a selection of emulators and simulators 22,33,34 for testing applications with pre-determined inputs. Similarly, Apple’s latest version of ARKit ships with a testing tool called “Reality Composer” 28, which allows developers to record and replay system inputs for testing. There are also many general-purpose tools available for graphics and systems-level profiling, such as NVIDIA’s “Nsight” suite of tools 17, the “apitrace” tool for OpenGL 1, Android’s system trace CLI tool 8, and Windows profilers for system events 7 and DirectX applications 24. While all of these tools can be helpful during development and initial testing phases, they are generally suitable only for in-lab testing, as their operations are quite resource-intensive. ARCHIE can supplement these efforts by providing additional lightweight support when an application has passed code quality testing in the lab and is ready for user evaluations in the wild.

Quality of Experience (QoE) testing. In addition to commercial tools, there has been research aimed at quantifying the quality of experience (QoE) that a tester might have while using a given application. Papers such as 43,64,85 propose frameworks to identify potential system bottlenecks that could adversely affect user experience. However, what these systems provide is raw data output; it is up to the researcher to determine the point at which the perceived latency for an application operation becomes insufferable to the user. ARCHIE can supplement these testing frameworks by providing temporally-aligned user feedback data to identify exactly when changes in system performance start to impact the user.

Cross-platform testing. Other general-purpose testing tools focus on supporting heterogeneous hardware platforms 74, software platforms 69, and embedded devices 58. However, these testing platforms do not address the user experience component of the testing process. ARCHIE can supplement cross-platform testing frameworks such as these by presenting users with different implementations of a given interface on a given device in order to identify whether any issues or preferences exist from a user’s perspective when porting applications to new environments.

8 CONCLUSIONS

In this paper, we presented ARCHIE, the Augmented Reality Computer-Human Interaction Evaluator framework. We demonstrated the prevalence and limitations of using only questionnaires and interviews for testing augmented reality applications in the wild, and showed how our feedback framework can assist developers in identifying and debugging UX issues with negligible impact to system performance. In the future, we would like to expand ARCHIE to include more formal usability testing.

REFERENCES


