LETTER

App-Level Multi-Surface Framework for Supporting Cross-Platform User Interface Distribution

Yeongwoo HA[†], **Seongbeom PARK**[††], **Jieun LEE**[†], *and* **Sangeun OH**[†a)], ***Nonmembers***

SUMMARY With the recent advances in IoT, there is a growing interest in multi-surface computing, where a mobile app can cooperatively utilize multiple devices' surfaces. We propose a novel framework that seamlessly augments mobile apps with multi-surface computing capabilities. It enables various apps to employ multiple surfaces with acceptable performance.

key words: *multi-surface computing, user interface distribution, mobile frameworks, code instrumentation*

1. Introduction

With the recent rapid development of IoT, smart devices with various shapes and sizes of surfaces have been released one after another. This trend can potentially change the way users interact with mobile apps. To be more specific, users can now collaboratively utilize the surfaces of multiple devices to interact with mobile apps, instead of being limited to one device at a time. Such a new paradigm is called *multi-surface computing*, and it can provide users with new user experiences (UXs) for multi-device use. For example, YouTube's app screen consists of a video user interface (UI) that shows a video stream and a list UI that shows a list of related videos. If these two UI elements can be displayed on different devices, a user can watch a video on full screen while easily browsing the list of related videos at the same time.

There have been several solutions to enable multi-surface computing, falling into *app-level* and *system-level* approaches. The app-level approach is to use customized apps developed for multi-surface computing, such as cross-device seamless video streaming (e.g., Netflix) and multi-user collaborative document editing (e.g., Google Docs). Yet, this approach requires a lot of engineering effort to develop such customized apps, so only a small number of multi-surface apps are commercially available currently. On the other hand, the system-level approach is to redesign existing mobile platforms specifically for multi-device environments. For example, both FLUID [1] and PRUID [2] are Android-based mobile platforms that support multi-surface computing by distributing UI elements of a single app across multiple devices. As another example, FLUID-XP [3] further extends FLUID to distribute UI elements even across heterogeneous platforms (e.g., between Android and iOS). However, these approaches fundamentally require significant modifications to the core parts of mobile platform internals such as UI management and rendering, making it difficult to deploy new mobile platforms on users' devices practically. This is because mobile industry companies leading the current mobile platform market, such as Google and Apple, tend to be reluctant to make significant modifications to the core parts of their mobile platforms to maintain product stability.

To go beyond their drawbacks, this paper proposes UI-Scissor, a novel app-level framework that automatically transforms existing legacy apps built for a single device into multi-surface apps. UI-Scissor can extend app functionality to allow individual mobile apps to distribute their UI elements across devices, enabling flexible multi-surface utilization for users. Such extension is accomplished by instrumenting the necessary code logic for UI distribution to individual mobile apps, thus our approach requires no modification to existing mobile platforms and no significant engineering effort for app developers to develop multi-surface apps. Furthermore, UI-Scissor aims to achieve the following design requirements. *i) App transparency*: The app transformation should be performed transparently to a mobile app's behavior. That is, the transformation should not interfere with or distort the app behavior unintentionally. To do this, UI-Scissor is designed to minimize the impact of the app transformation by providing most core functionalities as a separate background service and inserting only a minimal communication interface into a mobile app. *ii) Cross-platform compatibility*: The converted mobile app should be able to distribute UI elements across multiple heterogeneous devices in a platform-agnostic manner. To this end, UI-Scissor transforms a mobile app's UIs (i.e., native UIs) into appropriate web UIs that can run on different mobile platforms upon performing UI distribution.

2. Background

To concretely explore the app transformation for UI distribution, we target Android apps with graphical user interfaces (GUIs). The screen of an individual app consists of a collection of UI elements (e.g., buttons, text). From a user perspective, a UI element is the smallest unit with which a user can interact within an app. To represent these UI elements, Android's UI subsystem has two types of UI objects: *widgets* and *layouts*, which are managed in a tree structure

Manuscript received September 7, 2023.
Manuscript revised November 27, 2023.
Manuscript publicized December 19, 2023.

[†]The authors are with the Department of Software & Computer Engineering, Ajou University, Suwon 16499, South Korea.
[††]The author is with the Department of Digital Media, Ajou University, Suwon 16499, South Korea.
a) E-mail: sangeunoh@ajou.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2023EDL8060

known as the *UI tree*. A widget serves as a leaf node in the UI tree, representing a graphical component mapped to each UI element on the app screen. On the other hand, a layout acts as an intermediate node in the UI tree, functioning as a container that determines the positioning of its child nodes on the screen. In Android's UI rendering process, layouts first determine the position of each widget. Then, the widgets are drawn to their designated positions by utilizing their respective graphical states (e.g., text, color, image, etc.). After that, UI elements displayed on the app screen can be often updated by external events such as user inputs or incoming network data. For example, when a user presses a button UI in a mobile app, the user input is passed to the app logic part (i.e., event listeners). When processing the user input, the app logic part can modify the graphical states of specific widgets by invoking their local functions, resulting in the update of the corresponding UI elements. To enable UI distribution to multiple devices, UI-Scissor needs to ensure that the UI rendering and update handling process can be supported seamlessly in multi-device environments.

3. System Design

We present UI-Scissor, a novel multi-device framework that extends app functionality to enable mobile apps to distribute their UI elements to remote devices. Such extension is achieved by automatically adding a new layer to a mobile app through code instrumentation, which can minimize the engineering burden on app developers.

3.1 Workflow

As shown in Fig. 1, UI-Scissor supports multi-surface computing for mobile apps through *offline* and *online* phases.

Offline phase. An app developer transforms his/her legacy mobile app, which was initially developed for single-device environments, into a new multi-surface app by using App transformer. App transformer is a authoring tool provided by UI-Scissor and inserts additional code (called UI distribution layer) into the existing app. As an output, the developer can acquire the multi-surface app's package file (i.e., APK) and release it to app markets like Google Play. Then, End-users can easily download and install the multi-surface app to run the multi-surface app on their host device.

Online phase. On the host device, a multi-surface app (called a host app) is capable of distributing its UI elements to remote devices (guest devices) through three sub-steps. *i) Pairing*: Before running the host app, the end-user first

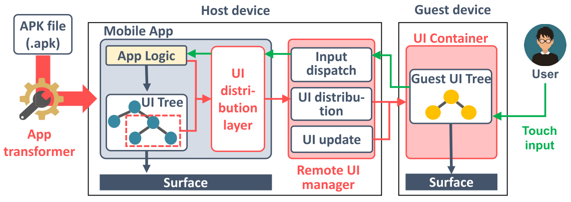

Fig. 1 UI-Scissor design overview

needs to register trusted guest devices with Remote UI manager. Remote UI manager is a background service provided by UI-Scissor and automatically establishes network connections with the registered devices when it detects that they are nearby. After that, the host app can seamlessly interact with the registered devices through Remote UI manager. *ii) UI distribution*: UI distribution layer injected into the host app provides an intuitive interface for selective UI distribution. The user can activate this interface with multi-finger tapping and select desired UI elements. Upon user request, UI distribution layer extracts the selected UIs from the host app's UI tree and delivers them to a guest device. On the guest side, UI container, which operates as a background service, displays the distributed UIs via local rendering (which we call *guest UIs*). *iii) UI interaction*: After UI distribution, UI-Scissor enables the user to interact with the host app through the host and guest UIs in the same way as if all UIs were on the same device. For example, if the user touches a button distributed to the guest device, the input event is forwarded to the host side and handled by the host app logic accordingly. If necessary, the host app can update some guest UIs' states.

3.2 Key Feature Design

UI-Scissor provides several key features as follows.

App transformation. App transformer extends app functionality by inserting additional code for UI distribution into a mobile app's APK file. A crucial design consideration is determining the appropriate code logic to inject. Rather than injecting all functions for multi-surface computing directly into the mobile app, which could lead to complexity and unexpected side effects, UI-Scissor minimizes injected code volume. It supports core functions by providing a separate background service (Remote UI manager) and injects only a minimal interface (UI distribution layer) into the mobile app to interact with the service. This design mitigates potential side effects. App transformer is implemented using *Soot* [4], an open-source static analysis library.

Cross-platform UI distribution. In UI-Scissor, UI distribution is initiated when the user completes selecting the desired UI elements. First, UI distribution layer extracts the graphical states of the selected UIs from the host app's UI tree and transmits them to the guest side via Remote UI manager. These graphical states indicate essential data for UI rendering, encompassing not only general properties (e.g., UI type, width, height, identifier) but also properties specific to the UI type (e.g., contents, font size for text UI). On the guest device, UI container decodes the received graphical states and reconstructs the distributed UIs. Here, one important issue is the challenge of *device heterogeneity*. In common multi-device environments, many devices may operate on various mobile platforms such as Android, iOS, and so on. Regarding this trend, the recreation of the host app's UIs (referred to as native UIs) may be infeasible if the host & guest devices are based on different platforms. This is because the different platforms internally are based on different instruction sets, API functions, and rendering mechanisms. To tackle this problem, we design UI container to generate web UIs from the graphical states of native UIs. Fundamentally, web UIs are built with web programming languages like HTML and Javascript, so they have a high portability that allows the same web UIs to work on various devices. To leverage the advantage of web UIs, UI container internally maintains the mapping table between native UIs and web UIs, which helps generate the necessary web UIs with the graphical states of the distributed native UIs. When the graphical states are delivered from the host side, UI container first identifies the type of each native UI. Then, it successfully creates appropriate web UIs with the received graphical states by referring to the mapping table according to the UI type. We implemented UI container using a Javascript-based React Native [5], which is a cross-platform mobile development tool.

UI state synchronization. Fundamentally, a mobile app updates UI elements by calling local functions, implying that distributed UIs also need updates when their local functions are invoked for multi-surface computing. App transformer injects *UI synchronization code* into locations where local functions of UI elements are called, ensuring correct multi-surface computing. The injected code requests UI distribution layer to modify the graphical states of a guest UI as if the local function is invoked on the guest side. Upon execution, UI distribution layer checks if the UI element involved in the function call is distributed and if so, passes modified graphical states to the guest device for UI container updates. One challenge is determining the graphical states modified by invoked functions. While UI functions in the Android API are well-documented for their impact on graphical states, custom UI functions defined by app developers are unknown and app-specific. App transformer addresses this by using class hierarchy analysis (CHA) to identify which graphical states are modified by each custom UI function before code instrumentation. CHA generates a call graph, linking call sites to possible class functions. This allows App transformer to identify functions invoked by custom UI functions, analyze their impact on graphical states, and guide UI distribution layer in updating the relevant states on the guest.

Discussion on the deployment of UI-Scissor. To scale the deployment of UI-Scissor, we propose a model where a server (e.g., an app market server or a dedicated UI-Scissor server) handles app transformation for multi-surface computing. App developers can submit their APK files to the server to add multi-surface features. The server transforms the APK files and provides the developers with the modified versions for release on app markets. Users can then download these multi-surface apps from app markets as they do with usual legacy apps. Note that this app transformation process hardly impacts server performance, typically taking only a few minutes, as demonstrated in Sect. 4.

4. Evaluation

We developed UI-Scissor to demonstrate seamless selective UI distribution across devices. Our evaluation utilized the ours implementation based on Android 11 and React Native 0.69. We employed two real commercial devices (Google Pixel 5 and 4a) and one virtual iOS device (iPhone 14) emulated on a MacBook Air. All devices were connected to the same Wi-Fi access point with a throughput of 433 Mbps and a round-trip time (RTT) averaging 3.43 ms.

Coverage test. To explore how well UI-Scissor supports potential multi-surface use cases, we extended ten legacy apps downloaded from Google Play. Subsequently, we launched each app on Google Pixel 5 (host device) and distributed several UI elements to two types of guest devices: one being an Android device identical to the host side but different model (i.e., Google Pixel 4a), and the other being an iPhone 14 emulator. Note that the emulator is enough to demonstrate that UI-Scissor can support UI distribution for iOS-based devices because it has the same software stacks as real iPhone devices. Table 1 displays the list of use cases employing legacy apps. We confirmed that UI-Scissor enables seven apps to successfully operate over multiple devices. Particularly, even in heterogeneous environments with iPhone 14 as a guest device, the legacy apps performed multi-surface computing seamlessly. This is because UI-Scissor can reconstruct proper web UI elements on the guest side, utilizing the graphical states of the legacy apps' native UI elements.

On the other hand, UI-Scissor cannot support three apps—Yanolja, Messenger, and Free Fire—for different reasons. First, even if our framework accomplished app transformation for Yanolja, this app failed to pass the signature verification process, resulting in its crashing. This is because we used an invalid signing key when repackaging this app during our experiment. This implies such issues will not occur when app developers with correct signing keys utilize UI-Scissor. Meanwhile, UI-Scissor could not conduct even app transformation for Messenger and Free Fire. This is because they leverage third-party UI frameworks (e.g., React Native or Unity) that manage all UI objects through internal data structures while not allowing UI-Scissor to access UIs. However, this is not severe limitation enough to hinder UI-Scissor's applicability, considering the prevalence of such third-party UI frameworks. We analyzed 50 legacy apps with the highest cumulative downloads from Google Play and observed that 13 of these apps utilize third-party frameworks. This indicates that UI-Scissor can still support the majority of apps, encompassing 74% of the apps we examined.

App transformation overhead. Table 1 shows the app transformation overhead, including 'Transformation time' and 'APK size'. UI-Scissor can augment UI distribution capabilities in less than five minutes, with the APK size increasing by under 10% after transformation. The minimal time/space overhead results from UI-Scissor's design, minimizing injected code in legacy apps. Notably, the camera app's APK size decreased by 16% due to Soot's removal of unnecessary code logic during app transformation.

UI distribution time. To assess UI-Scissor's performance in distributing UIs to the guest device, we measured UI distribution time for each use case scenario. This time is defined as the difference between when a user triggers

Table 1 The result for coverage test. 'Transformation time' indicates how long it takes to transform a legacy app into a new multi-surface app, and 'APK size' indicates the APK size of the generated multi-surface apps. Yanolja is a hotel reservation app and Free Fire is a mobile shooting game.

Use case scenario	App name	UI type	Android	iOS	Transform-ation time	APK size (increase ratio)	Distribu-tion size
Display calculation results on a guest device	Calculator	Text	O	O	42s	7.3MB (5.6%)	0.19 KB
Edit text on a guest device	Notes	Edit Text	O	O	48s	8.9MB (6.8%)	0.17 KB
Sharing images to a guest device	Gallery	Image	O	O	1m 45s	18.7MB (6.5%)	35.07 KB
Control a camera with a guest device	Camera	Button, Image	O	O	53s	9.2MB (-16%)	3.68 KB
Control flashlight with a guest device	Flashlight	Button, Image	O	O	44s	8.5MB (5.8%)	13.84 KB
Edit a schedule with a guest device	Calendar	Edit Text, Button, Image	O	O	48s	8.9MB (7.7%)	1.75 KB
Control painting tools with a guest device	Draw	Button, Image	O	O	31s	7.3MB (6.4%)	5.88 KB
Make hotel reservation with a guest device	Yanolja	Edit Text, Button, Image	X	X	4m 20s	538.9MB (7.5%)	-
Write a chat message with a guest device	Messenger	Edit Text, Button	X	X	-	-	-
Control game with a guest device	Free Fire	Button	X	X	-	-	-

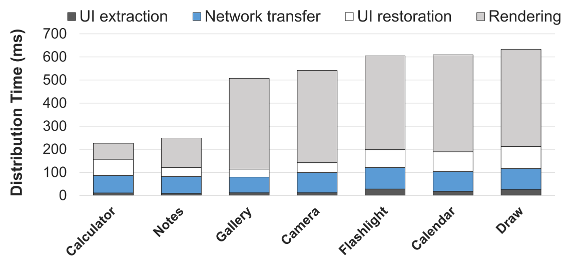

Fig. 2 UI distribution time according to seven legacy apps.

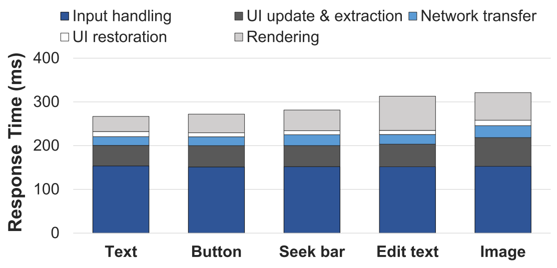

Fig. 4 UI response time according to five UI types.

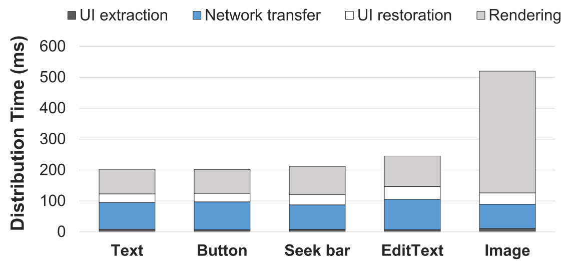

Fig. 3 UI distribution time according to five UI types.

UI distribution and when the guest device's screen is last updated. Figure 2 breaks down the UI distribution time for the seven legacy apps. *Calculator* and *Notes* have UI distribution times of less than 250 ms, suitable for interactive use. However, other apps exhibit longer times, ranging from 507 ms to 633 ms, primarily due to their long rendering time. This is because these apps' UIs contain Base64-encoded images. These data are known to cause high rendering overhead within React Native used by UI container [6]. To detail this, we measured UI distribution times for five different UI types, as shown in Fig. 3. It reveals that only image UI incurs high rendering times, while the rest are quickly distributed with short rendering. However, UI distribution is a one-shot overhead for supporting a multi-surface scenario, and the overhead for image rendering can be reduced by using optimizations provided by React Native, such as the native module feature. Applying these optimizations is left for future work.

UI response time. We measured UI response time for updating the five types of guest UIs. UI response time is defined as the time difference from when a user gives touch input on the guest screen to when the guest screen shows the result of the touch input through each guest UI. As shown in Fig. 4, UI container takes about 153ms to process the user input, an inherent overhead caused by Native React. UI-Scissor efficiently handles UI update operations, resulting in response times ranging from 267ms to 321ms—moderate for interactive use. Note that the rendering time for UI updates is smaller than for UI distribution because React Native creates a new data structure (i.e., virtual DOM tree) for rendering upon UI distribution, but during UI update, it only modifies some graphical states of the previously generated data structure and re-uses it for rendering.

5. Conclusion

This paper presents UI-Scissor, a novel framework that automatically transforms single-device apps into multi-surface apps. Our framework provides *i)* an authoring tool that augments existing legacy apps with UI distribution features and *ii)* services that help multi-surface computing. We expect UI-Scissor can accelerate the development of multi-surface apps, providing novel user experiences.

Acknowledgments

This work was supported by the NRF grant (NRF-2021R1F1A1063785).

References

[1] S. Oh, A. Kim, S. Lee, K. Lee, D.R. Jeong, I. Shin, and S.Y. Ko,

"FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction," MobiCom, vol.23, no.4, pp.25–29, 2019.

[2] M. Cui, M. Lv, Q. He, C. Zhang, C. Gu, T. Yang, and N. Guan, "Pruid: Practical user interface distribution for multi-surface computing," DAC, pp.679–684, 2021.

[3] S. Lee, H. Lee, H. Kim, S. Lee, J.W. Choi, Y. Lee, S. Lee, A. Kim, J.Y. Song, S. Oh, S.Y. Ko, and I. Shin, "FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience," MobiCom, pp.762–774, 2021.

[4] Sable Research Group, "Soot - a java optimization framework." https://github.com/Sable/soot.

[5] Meta platforms, Inc., "React native - learn once, write anywhere." https://reactnative.dev.

[6] Stack Overflow, "React native image loading is too slow." https://stackoverflow.com/questions/65727698/react-native-image-loading-is-too-slow.