A Case Study of Combining Two Cross-platform Development Frameworks for Storybook Mobile App

Beomjoo Seo
School of Games, Hongik University
Sejong 30038, South Korea
[e-mail: bseo@hongik.ac.kr]

Abstract

Developers often use cross-platform frameworks to create mobile apps that can run on multiple platforms with minimal code changes. However, these frameworks may not suit all the needs of a specific app, so developers may also use native APIs to add platform-specific features. This method eventually dilutes the advantages of cross-platform development methodology that aims to reduce development costs and time, and often leads to a decision to return back to the original native mobile development methodology. In this study, we explore a different approach: combining different cross-platform tools to develop a storybook mobile app that meets various requirements. We have demonstrated that integrating two cross-platform solutions can be used reliably to develop complex mobile applications. However, we also report that this approach can introduce unforeseen issues such as sandbox redundancy, unexpected functional burdens, and redundant permission requests. Despite these challenges, we believe that combining two cross-platform solutions can be applied to a variety of functional and performance requirements, enabling the development of more sophisticated mobile applications at lower costs and with shorter development timelines than traditional mobile app development methodologies.

1. Introduction

Mobile app development companies strive to discover compelling ideas quickly, prototype proof-of-concept apps, and implement and launch the apps in the fast-faced mobile marketplace. However, they frequently encounter challenging implementation issues during the development process. Small and medium-sized companies with limited financial resources face significant challenges in allocating budgets and manpower capable of handling all the different development environments offered by individual mobile platforms.

Mobile applications can be developed using three different approaches: native, hybrid, and cross-platform solutions. The native approach involves building applications for specific platforms like iOS and Android using platform-specific programming languages such as Swift or Java. This approach offers higher performance due to the direct use of native features and APIs, but it often results in longer development time and higher costs.

In contrast, the hybrid approach utilizes web technologies like HTML, CSS, and Javascript to render an app within a native container like WebView. This allows for deployment on multiple platforms, with lower development costs and faster development time than the native approach. However, it cannot fully utilize all available native features, leading to comparatively lower performance.

Finally, the cross-platform approach builds apps using a single codebase that can be deployed on multiple platforms. It falls in between the native and hybrid approaches with lower development costs and faster delivery time than the native approach, but higher cost and slower delivery time than the hybrid approach. It also has limited access to native features, and its performance falls between the other two approaches.

Apart from the above approaches, Progressive Web Apps (PWAs) offer an alternative solution that creates web apps that can be deployed through the web without the need for any app marketplaces [1]. While PWAs use the same web technologies as the hybrid approach, they have much cheaper and shorter development costs and time, and can provide app-like functionalities even offline, without significant quality degradation.

Nowadays, No-Code/Low-Code solutions are gaining more attention for specific targeted mobile apps, as they allow non-technical users to create applications without requiring extensive knowledge of programming languages [2].

While these solutions can lower the entry barrier for developers, reduce development costs, and speed up development time, many cross-platform solutions are widely used in development environments that require medium-to-high complexity of user requirements or high-performance demanding. According to a 2021 survey by Statista, roughly one-third of mobile developers use cross-platform technologies, while the rest still use native approach [3].

As cross-platform development methodologies have matured, they are getting closer to providing user experience similar to that of native tools in terms of performance. However, each cross-platform solution still has specific performance weaknesses in certain areas. Companies that had actively adopted a certain cross-platform solution have reverted to native approaches due to these vulnerabilities [4]. In this study, instead of switching back to native solutions, we investigate whether integrating multiple cross-platforms can overcome these issues by maximizing the benefits of a small set of single codebases for multi-platform environments while tailoring to the strengths of each platform. We examine the effectiveness of our proposed method in developing an interactive storybook mobile application with varying performance requirements.

This article is organized as follows. In Section 2, we review various cross-platform development and image object extraction methodologies, which are essential concepts for our targeting storybook mobile app. Section 3 introduces the service architecture of a mobile app developed for this research, outlines the app's requirements, and describe the approach used to meet these requirements. In Section 4, we explain how image object extraction and gesture-based image editing techniques, which are typically heavily dependent on native features, were implemented in a cross-platform solution and integrated with another cross-platform environment. In Section 5, we evaluate the mobile app's real-time capabilities and explore the reliability of the app's complicated tasks. Finally, Section 6 summarizes the study's findings and presents our conclusion regarding the proposed approach.

2. Related Work

2.1 Cross-Platform Development

Biørn-Hansen et al. classified various cross-platform frameworks into five categories by their development approaches: hybrid, interpreted, cross-compiled, model-driven, and progressive web apps [5]. The hybrid approach uses WebView to display web contents and interact with web applications [6]. Since the WebView includes all internet-related browsing functionalities and is supported by all mobile platforms, mobile app developers can embed web technologies into the WebView to add new UI-centric business logic. However, due to its suboptimal performance output, the hybrid approach is not a popular choice for developing complex and computationally intensive mobile applications.

The interpreted approach, such as React Native, allows developers to create mobile applications using the Javascript language [7]. The apps are then deployed to devices with a platform-specific Javascript interpreter, which interprets and executes the Javascript code on-the-fly to render native user interface components on the device display. To communicate between the Javascript code and native interface layers, it uses a bridging technique, which can result in performance bottlenecks for apps utilizing this approach.

In contrast, the cross-compiled approach, exemplified by Flutter [8], Xamarin [9] and even Unity3D [10] or Unreal [11] game engines, on the other hand, compiles a common language into native byte code, resulting in near-native performance and faster app performance. However, it may require more development effort and skill than the interpreted approach.

The model-driven approach generates an app from templates, pre-built models, or specific description languages, abstracting all platform-specific details when building user interfaces and application logics, and does not require knowledge of platform-dependent programming languages. Many no-code/low-code solutions belong to this category.

Finally, progressive web apps (PWAs) approach provides users with a web app that looks and feels similar to a native or cross-platform built app. While PWAs-powered apps can be downloaded to run in offline mode, they cannot fully utilize all features of the underlying mobile platform and may not meet performance requirements.

Companies developing mobile apps choose the most appropriate development solution, such as cross-platform, native, or web app solutions, based on the requirements of the mobile app they want to develop. Despite the maturity of their understanding and application of these technologies, there are often situations where previously utilized solutions are no longer suitable for evolving mobile app requirements. In such cases, companies must decide whether to invest in their existing solutions to meet new requirements or transition to a new development environment that better fits their needs.

This study seeks to explore whether it is possible to apply a customized development approach that selectively chooses a different development environment that fits specific requirements while retaining the existing development environment. To achieve this, we will conduct a case study of integrating React Native development environment, an interpreted approach that has a broad language user base and specializes in mobile UI/UX development while providing various native modules as needed, with Unity3D, a cross-platform development that is widely used in mobile game development with a large developer base.

2.2 Image Matting

Our storybook app can provide mobile users with the ability to directly change the existing contents of a story page. To extract a desired object from a camera in real time, we use digital matting solution embedded in our target mobile application.

Matting (also widely termed as green/blue screen keying or chroma-key filtering) is a method of separating a foreground from a rectangular background image and then synthesizing the foreground image with a desired background. The technique has been popularly used in various fields such as motion pictures and live streaming. Generally speaking, the separation has been a challenging problem. As the simplest solution to this problem, blue screen matting (or interchangeably termed, chroma key processing) has been widely used by placing a fixed color (usually green) in the back screen to quickly extract foreground objects of interest, and simultaneously substituting with a new background image [12]. Since the chroma key technique is too sensitive to lighting, it is error-prone when shadows are cast that you need to work in an environment with adequate lighting and a plain background.

Recently, the image separation technique has been evolved to accurately identify the background of salient objects from an image using deep learning [13]. Google employs the use of web-based machine learning technology to blur or replace existing background surroundings in its video conferencing application, Google Meet. Its segmentation model down-samples an incoming video frame to a low-resolution image, then feeds the image to the segmentation model to find a segmentation mask, and finally recovers the mask as fine as possible. The segmentation model is based on MobileNetV3-small for encoding blocks, which reduces the model size by 50% at the expense of a small loss in weight precision, resulting in 193K parameters and 400KB in total size [14]. After the segmentation, either bilateral filtering to smooth the segmentation mask or light wrapping for mixing the foreground and replaced background images is applied.

As the deep learning-based image segmentation technology matured, it was immediately applied to mobile devices and start to be used in everyday life. Apple recently announced a 'Visual Look Up' running on iOS 16 that allows users to tap a subject not only in an image but also in a video frame and life it from the background. To support this, it uses the latest hardware called Apple Neural Engine, which is optimized for energy-efficient execution of deep neural networks on Apple devices, and a software packaged called HyperDETR, a panoptic segmentation integrated detection transform framework to support higher output resolution and more region proposals [15].

In this study, we aim to apply the chroma key shader-based object extraction method in a cross-platform development environment, which is relatively easy to implement compared to other methods, yet has a somewhat high implementation complexity that utilizes the GPU performance embedded in mobile devices as well as various shader functions. Additionally, we seek to find alternative solutions to address these problems.

3. Design of Storybook Mobile App

This section outlines the fundamental structure of the content service for creating, distributing, and accessing fairy stale storybooks. It also covers the usage scenario of the fairy tale mobile app, which is integrated with this service, along with the mobile app's requirements to support it. Additionally, we provide a detailed analysis of how these requirements can be fulfilled.

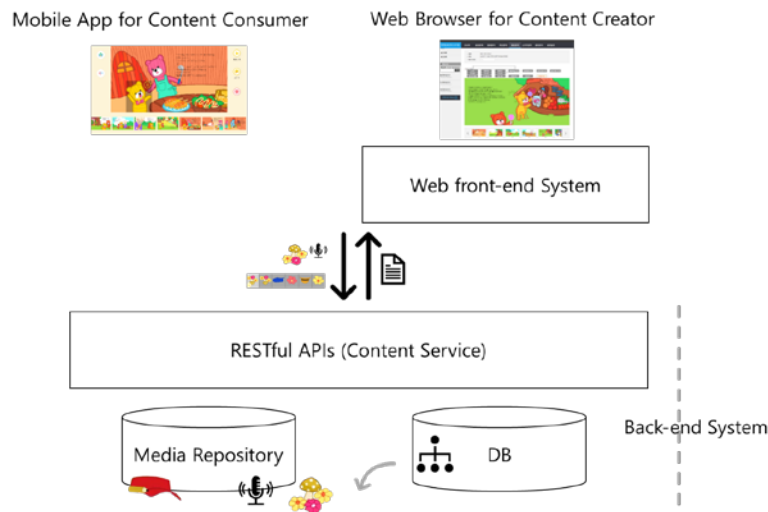### 3.1 Overview of Storybook Content Service

Fig. 1. System overview of storybook content service.

Our web-based storybook content service is based on a traditional client/server communication model as depicted in **Fig. 1**. Users of this content service fall into two groups: content creators or content consumers. Content creators can upload high-quality raw media data to the system and organize them to create electronic storybooks. Then they distribute a newly created storybook via a proprietary storybook marketplace and sell value-added assets such as sticker images, animatable contents, and narration audio files or background music files that help content users fully engaged in the storybook world. Content consumers, on the other hand, search interesting storybooks from the service, read them, and customize them as the way the content creators intend. The creators can specify text or image areas within e-book pages, where the consumers change at will.

The web-based server system consists of two systems: a back-end system and a front-end system. The back-end system manages all resources, including high-quality images, audio files, and storybooks. Media data is stored on storage devices and its file paths are indexed in a DBMS. The front-end system is a web-based storybook editing and publishing app that allows content creators to create, search, read, edit, and publish. In contrast, the mobile app component targets content consumers and allows them to purchase, play a storybook, and edit specific areas of the book where the content creators permit replacement with a photo.

Storybooks created through the content service are stored in tuple format in the server's DBMS. To transmit and deliver this information to users in real-time, a special transmission format is required. For general e-books, standardized document formats include PDF, EPUB, MOBI, and others.

However, PDF is designed for static documents and lack supports for complex animations or interactivity. While it is possible to embed some image animations, their capabilities are limited, and separate software is required to view and read them. On the other hand, EPUB or MOBI is a more flexible format than PDF and supports basic interactivity and animation such as page transitions [16]. However, they are still limited in their ability to display complex animations and lack the native mobile-level interactivity required for mobile content.

To enable complex image animation capabilities on a page-by-page basis in a mobile environment, it is necessary to develop a native mobile app or use a specialized format such as HTML5. When developing in HTML5, it is recommended to utilize SVG animation methods, but using CSS3 animations to drive SVG animations can be a performance penalty in mobile environments. To this problem in a cross-platform environment, a way is needed to parse the methods specified in SVG animation and programmatically translate them into an animation library that provides optimal performance on mobile.

Fig. 2. A sample storybook in JSON format and its sample page parsed as a JSON object.

This study utilizes a proprietary file format expressed in JSON format (as shown in **Fig. 2**) when collecting information stored in the DB to support the required complex image animations for storybooks in a mobile environment. When a user requests to retrieve a storybook, the backend system retrieves all the related object tuples from a database, builds them on the fly as a JSON file, and sends it over the network. The newly generated JSON files of the storybook are then stored in media storage and is served on subsequent retrieval requests. All content related services are served as Representational State Transfer web services [17], which the storybook authoring/publishing web app and the mobile app use to communicate.

3.2 Mobile App User Scenario

In this subsection, we will provide an overview of the user scenario for the mobile app, which targets content consumers in the service. **Fig. 3** depicts a schematic user flow design of the mobile app.

When a mobile user accesses the app, they can either sign up for membership or use the service without any user registration as an anonymous user. Free storybook content can be enjoyed by anonymous users, but certain features, such as image substitution or storybook personalization, are only available for registered users.

After signing up or logging in, the user can search for any storybooks available on the service and purchase them if they are not free. The user can choose to manually read the pages or play them automatically, with smooth transitions between pages and narrations. During manual reading, the user can swipe back and forth to switch pages or use the thumbnail list of all pages to navigate to a specific page. While playing the pages automatically, the user can play, pause, resume or rewind playback by swiping at any time. All animatable images are displayed during playback, synchronized with the original content creators' intentions.
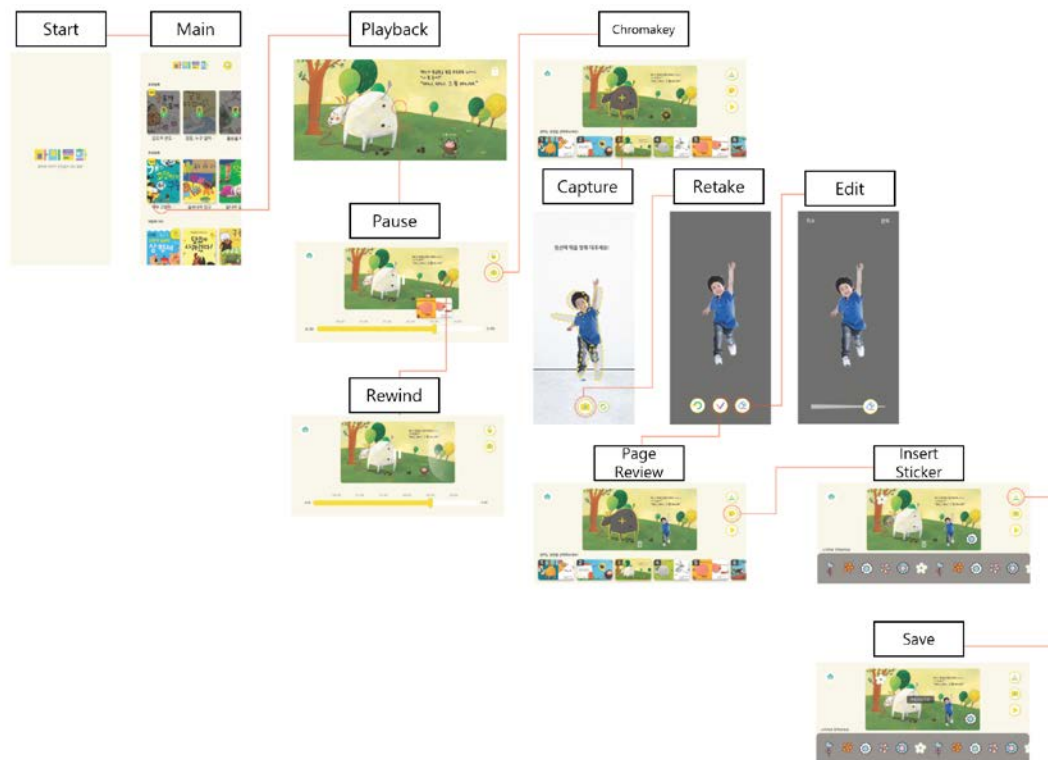
Fig. 3. Schematic user flow diagram for a storybook mobile app.

The mobile app also allows users to enter into an editing mode by tapping the screen while reading or playing. In this mode, users can purchase sticker images or stick image packs from the content marketplace and place them on any pages. The sticker images can be resized by multi-touch gestures. Some sticker images that are marked as animatable are animated after placement. After editing, the user can preview the page or play back from the page to ensure that everything was placed as intended.

Certain pages permit users to replace images as authorized by the content creator. In such pages, users can tap on the image to be replaced and enter the image substitution mode, which shows a real-time camera preview. The user can extract an object of interest from the preview camera screen by applying chroma key filtering shader and may change the chroma key color.

To change the chroma key filtering color, the user can long-press on the camera screen to recognize colors in the preview screen and receive immediate visual feedback for the newly chosen chroma key color. Users can even choose multiple chroma key colors, applying them on the camera screen in real-time to ensure that the intended image object is correctly recognized. Once the object extraction is complete, the user can erase unwanted areas from the captured image or recover the desired area by touch-drawing. To pinpoint the desired area, the user can zoom in or out. Once the user has completed these actions, they can return to the storybook page to view the newly replaced image.

The mobile app allows users to maintain multiple edited versions of the same storybook. Users can also replace some of the wordings in the storybook, changing the heroine's name to a loved one. Additionally, users can order offline printing of their personalized storybook through the mobile app.

3.3 Mobile App Requirements and Analysis

The functions of a storybook mobile app, depicted in the user flow diagram discussed in Section 3.2, can be broadly categorized into four groups: general UI/UX-driven interactive tasks, storybook playback, storybook page editing, and image object extraction. The UI/UX related tasks consists of common mobile-based UI/UX components such as managing user authentication, searching for storybooks and stickers, purchasing for storybooks and stickers, managing purchase history, and accessing offline printing request services. These tasks are relatively easy to implement because many cross-platform development environments are designed to provide UI/UX-centric business logic effortlessly and seamlessly. They are often organized into discrete screens, allowing for seamless navigation between screens and displaying dialogs using a variety of UI transition effects. Thus, this discussion focuses on the requirements and challenges of the remaining three tasks that are not UI-related.

Storybook Playback

Developing a storybook player in a mobile environment is a complex task, similar in difficulty to create an electronic book player. The player must handle multiple types of multimedia content, including images, audio, and text, and synchronize them in real-time. Coordinating a number of animations between these content types can be particularly challenging, especially on low-end devices with limited processing power and memory. Additionally, supporting various screen sizes and resolutions makes it challenging to optimize the player for all devices.

To provide a seamless user experience, the player must prepare page scenes in advance to ensure smooth transitions from one scene to another. This requires effective memory usage management, especially when dealing with large numbers of high-resolution images. The player should also enable fluent navigation back and forth based on user requests, without causing disruptive experiences.

Some of these challenges cannot be solved solely by the mobile app and may require server-side help, if applicable. Therefore, developing a robust and user-friendly storybook player in mobile environment demands a combination of technical expertise, UI/UX design skills, and careful consideration of user needs and expectations.

Storybook Page Editing

While web-based authoring tools offer the ability to edit and compose the entire page from scratch, mobile apps provided limited editing features such as attaching or detaching stickers. In edit mode, this feature seamlessly integrates with page playback, allowing users to review the current page at any time or ensure uninterrupted playback of the entire storybook. For

attaching images, multi-touch gestures can be used to smoothly move them to the desired position and make fine adjustments like zooming or scaling. Additionally, it enables users to move specific layers within the stacked images on a page, altering the layering layout. Thus, this feature combines page playback, diverse image movement, and transformation capabilities, leveraging external storybook playback functionality. On the editing screen, typical UI tasks include loading stickers from a list, selecting and placing them, and navigating to another page for editing. On mobile devices, smooth real-time processing of image detachment, replacement, and touch interactions is crucial.

Image Object Extraction

In this system, the storybook creator can place image objects on certain pages that mobile consumers can modify. These customizable images are referred to as 'interactive' images in this research. In the page editing mode, consumers can view and interact with the interactive images, and they can enter the image selection mode by tapping on an interactive image. In the editing mode, interactive images (see **Fig. 4(c)**) are accompanied by guide images (see **Fig. 4(b)**) that illustrates the available actions or transformations. Users can select an interactive image to proceed with replacing the original image with a new one. The replacement image can be chosen from the mobile device's photo gallery or captured directly using the camera. While the former is a common UI task, this research primarily focuses on designing the functionality for direct image extraction using the camera.
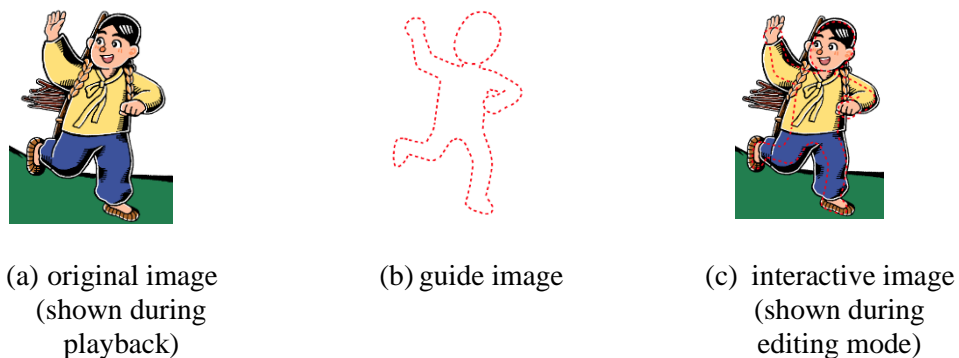

(a) original image
(shown during
playback)

(b) guide image

(c) interactive image
(shown during
editing mode)

Fig. 4. A sample interactive image demonstrating the combining an original image with its guide image.

When the user taps on an interactive object, the mobile app enter the camera preview mode. A dotted guide image associated with the interactive object appears on the camera screen, assisting the user in achieving the desired pose. During this time, the user can visually preview the extract image object result by displaying the shooting target separated from the background in real time through chroma key filtering. Furthermore, the user has the option to manually adjust the brightness, saturation, and color information relevant to the default chroma key filtering directly on the camera screen, allowing them to immediately assess the level of separation achieved from the background. The process of modifying the chroma key filter information should be straightforward and intuitive. With the real-time processing capability, the user can enhance the extraction quality by either repositioning the subject against a background that is easier to extract or selectively removing challenging background objects from the rest of the composition.

The process of manually removing unwanted portions and restoring desired portions from the background, as well as modifying the real-time bitmap image after chroma key processing, demands a high level of responsiveness and real-time capability, especially when performed through multi-touch interactions.

4. Implementation

This section presents the implementation methodologies employed in the storybook mobile app. Prior to delving into the specific details, let's first explore how the storybooks are managed on the server and transmitted to the app through the network.

For the development environment, we utilize Bitnami WAMPSTACK, which provides a comprehensive platform. Once the development is completed, we deploy the server code to Cafe24, a popular web-hosting service in Korea. Through this hosting service, we utilize the entire LAMPSTACK software suite, including Apache2, PHP, and MariaDB.

4.1 Integrating Development Environments

In this study, we explore the utilization of various cross-platform solutions to improve performance by creating modules that optimize specific functions within each solution. By combining these modules, we aim to reduce development time while meeting functional requirements effectively.

To minimize additional burdens such as increased operational cost due to interoperability with individual tools, setting up development environments, and developer availability, it is crucial to limit the number of cross-platform development tools to two widely adopted models in the industry. Based on this rationale, our study focuses on two primary development tools: React Native, a widely used framework for mobile app development, and Unity3D, known for its exceptional rendering performance and suitability for game development.

React Native, built upon React technology, is an open-source web framework developed by Meta. As a cross-platform solution, React Native focuses on improving responsiveness of native modules supported by mobile operating systems through JavaScript codes communicated via bridge channels. When a React Native application is launched on a mobile device, the operating system initiates a UI thread responsible for handling the user interface. This UI thread spawns a Javascript thread and a shadow thread. The Javascript thread processes the developer's code and delivers it to the shadow thread, which computes the layout and passes it to the native side through bridge communications.

On the other hand, Unity3D is specifically designed for game development, enabling developers to create applications with interactive gameplay, high-quality graphics, and efficient performance. It supports multi-platform development, allowing applications to be built for various platforms such as mobile devices, desktops, consoles, and even virtual reality and augmented reality devices. With its built-in editor, Unity3D facilitates the easy import and management of diverse assets, simplifying overall development process.

In this study, we have selected the React Native development environment as the primary cross-platform tool for the mobile application development. Given that user interface and user experience are critical components of general mobile apps, it is crucial to choose a development tool that facilitates rapid prototyping of various UI/UX effects. Consequently, we opted for the widely adopted cross-platform tool available at the time.

During the analysis of functional and performance requirements for our content service, our primary focus was on evaluating how well the React native platform supports these features. In cases where the React Native platform falls short in meeting the requirements, we

explore alternative solutions utilizing Unity3D.

To integrate these development environments, we employed the method proposed by Francois Beaulieu [18]. This approach involves building a module within the Unity3D environment and exporting it to the React Native development environment. Once exported, React Native recognizes the Unity3D module through Unity View [19], which is a plugin library facilitating communication between React Native and Unity3D modules via a dedicated bi-directional channel. Consequently, all functionalities are driven by the React Native main thread, with the Unity module being controlled by the main thread through the established communication channel. From the perspective of React Native, Unity3D modules are treated as conventional native modules. For a visual representation of our integration model, please refer to **Fig. 5**.
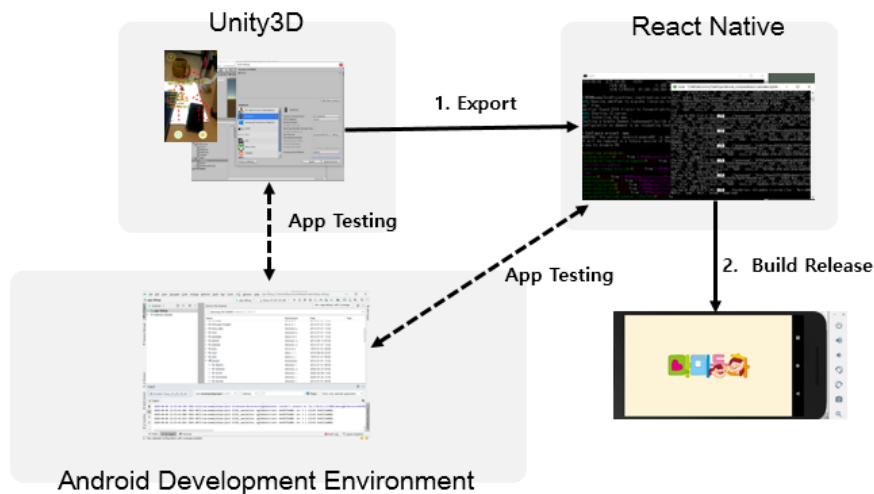
Fig. 5. Plugging a Unity Module to the React Native Development Environment.

Our integration approach, however, presents several challenges. When working with two different environments that operate as isolated sandboxes, there is a potential for redundancy in functionality implemented within both environments. For instance, in a scenario where all sandboxes are managed through server-established sessions, each sandbox utilizes separate session information due to their isolated web connections. This leads to complications in server-side processing. To address this issue, we designated a single communication agent within the React Native environment and established communication through it, rather than maintaining separate network agents. However, we observed that this solution introduced additional communication overheads, resulting in a more complex implementation of business logic and increased response time delays. Furthermore, the presence of two separate sandboxes leads to independent access permission requests from the mobile operating system. While efforts have been made to integrate these independent permission requests, a definitive solution has not yet been developed.

4.2 Storybook Playback and Page Editing

These two functionalities require a common feature of converting a JSON-formatted storybook into UI components. The pages within the storybook consists of various elements (text, audios, images, animation information, etc.) organized hierarchically and rendered sequentially. This means that earlier elements can be overlapped by later elements during

rendering. Individual elements are converted into Native UI components provided by React Native, forming a single page screen.

In the edit mode, after arranging all individual elements, new objects such as stickers can be inserted, deleted, or modified in specific orders within the page. These newly inserted objects should be continuously changed on the screen through drag-and-drop, while maintaining the overall layout of the page. The overall page structure remains unchanged, while specific parts are animated using the animation library provided by React Native. To achieve smooth animations, React Native adopts a preloading mechanism where the necessary animation information is sent to the native thread in advance, eliminating the need for continuous information exchange between the Javascript thread and the native thread. This approach is designed to avoid performance degradation. In this study, we utilized the Reanimated library to effectively use animation code in React Native [20].

In playback mode, all UI elements and media elements arranged on the screen are played according to the animation timing information. Different animation elements are synchronized with other media elements and performed sequentially or in parallel. Since React native preloads animation-related information to the native thread, the animation performance is proportional to the performance of the native thread, achieving native-level performance. Performance-critical aspects, such as audio playback or SVG animations like Lottie file playback, can significantly reduce the UI workload through the use of dedicated libraries [21]. Notably, the simultaneous playback of background music and narration audio, which requires playing two or more audio files simultaneously, is only possible within the media library provided by Expo, as other audio-related libraries fail to provide this functionality.

Among the playback and editing features, the most performance-sensitive part is animation combined with multitouch. In React Native, off-loading is applied to maintain such performance at the native level, using a third-party library. This enables the mobile app to provide complex storybook playback and editing features without performance degradation. Factors directly influencing this performance improvement are addressed through appropriate utilization of third-party libraries and other techniques during the development of the mobile app.

An important consideration in these features is the potential for memory resource shortages due to excessive use of image objects. Given the nature of storybooks, rendering numerous high-resolution images is often necessary for constructing a single page. Due to the high demands for image rendering, the mobile app frequently experienced memory shortages. Additionally, since the playback and editing of storybooks are not limited to a single book but may involve multiple storybooks, efficient management of images and proactive caching policies are necessary. In this regard, a server-side approach is employed, where images are prepared in various resolutions for different mobile devices, and the most appropriate image files are transmitted to each device [22]. Furthermore, instead of preparing individual images separately, an approach that combine non-overlapping images into a single composite image is applied, whose detailed descriptions are beyond the scope of this paper. To accommodate the use of a large number of image files, various solutions, such as appropriate image data indexing, and caching, are implemented through an independent image caching mechanism, reducing the memory requirements and loading time of the mobile app.

4.3 Image Extraction and Editing

Image Object extraction and editing is the key feature of our mobile app. **Fig. 6** depicts this process into six steps. The process begins with selecting an interactive item from a page edit mode (Step 1). After the selection, the app enters into a photo-taking task. The task can be

divided into two sub-stages: chroma filtering stage (Step 2 and 3) and editing stage (Step 4 and 5). The chroma filtering stage starts with displaying a red-dotted guide line on a camera surface (Step 2). Using the guide image, the user places objects of interest within an imaginary guideline rectangle. Here, the default chroma key background color is automatically detected and the image frame from the camera preview is applied by the chroma key filter and rendered in real time. Users can change the chroma keying color by long-pressing the screen. The app collects the target colors from a small area the user presses, averaging them to a single value, and immediately applies the average value to the newly selected chroma key (Step 3).
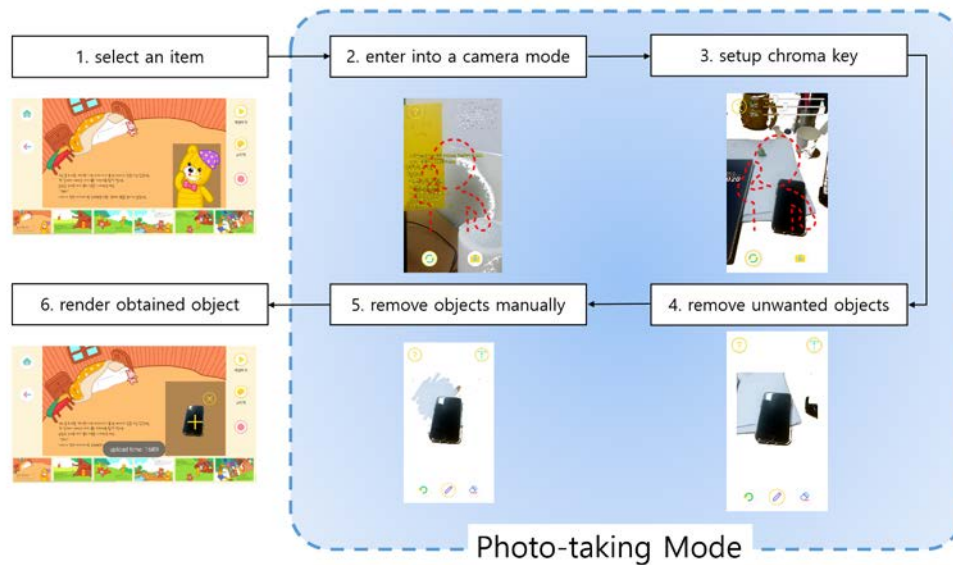
Fig. 6. Image Extraction and Editing Process using Chroma Keying.

The editing stage begins when the user takes a picture while cropping the surrounding area from the current chroma-filtered video frame (Step 4). Cropping an area in the frame is intended to reduce the image size to a smaller one, but the cropped area can be restored during post-processing if necessary. Before returning back to the scene, user can remove or restore crop region with a simple manual finger touch (Step 5). When all final touches to the crop image are done, the app saves all the resulting images (a high-resolution video frame and its chroma-filtered edited image) and return to the previous storybook scene with the freshly replaced image (Step 6).

The two stages can be implemented as a single operation or two separate operations, depending on the difficulty of implementation. Although they appear to have the similar requirements in terms of performance, there are distinct differences in the resource utilization involved. The former is more GPU-intensive because it focuses more on real-time chroma filtering (usually using a chroma-key shader), while the latter requires higher CPU and memory utilizations for instance pixel-level image conversion based on the user's touch movement.

Implementing these two stages in the React Native environment presents significant challenges. While it is possible to write shader code in WebGL format for chroma filtering in React Native, it requires additional mandatory components such as react-native-canvas and react-native-webview. However, our preliminary attempts using these components have reveal ed that real-time touch and rendering are not feasible due to certain limitations. The touch

events, initiated by the user, are first recognized by a native module and then delivered to a React Native Javascript thread to execute developer specified tasks. After the execution, the results return back to the native module through a communication channel, as depicted in **Fig. 7**. This process causes significant lags and jagged animation effects due to the continuous occurrence of touch events during drawing. Unlike the React Native animation library, there is no way to offload developer's code to the native thread, and all React Native canvas-related libraries face the same problem. Additionally, a single gesture involves a processor-intensive bitmap Blit operation [23], which modifies nearby image pixels in real-time around the touch area.
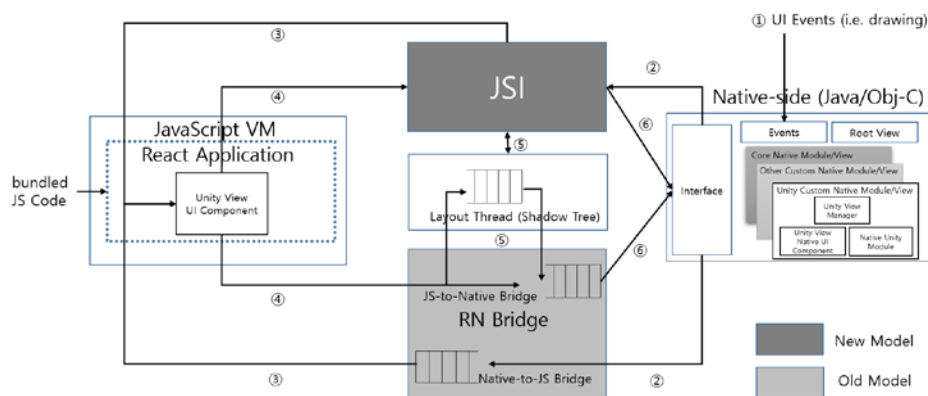

Fig. 7. UI Event Delivery Path on React Native.

These observations are primarily a result of the React Native architecture. To address this issue, the React Native maintenance team has recently introduced the JavaScript Interface (JSI) and changed the communication model between the main thread and the native thread. The new model simplifies the communication model from asynchronous to synchronous operations but still incurs significant communication overhead between the main thread and the native thread.

To overcome these limitations, we decided to use the Unity3D solution for the photo-taking task. Unity3D is well-suited for image manipulation and editing, and it eliminates the need for communication between the native thread and the React native main thread.

In Unity, the chroma key processing logic was implemented using the ChromaKey/Unlit/Transparent shader based on the uChromaKey shader asset, which is provided as open source [24]. By leveraging shaders, it becomes easier to replace them with a better one, simplifying the refinement of image extraction logic. The chroma key color can be modified through communication between the shader and the app. Additionally, the Unity graphics library provides the Blit function, which enables reading, removing, or applying a specific chroma key color to designated areas of a pixel image loaded onto a RenderTexture. During the editing process, operations such as removing desired pixel regions or restoring them can be implemented using the libraries provided by Unity. These various pixel-level image manipulation tasks can be accomplished using the functionalities offered by Unity. It is important to note that such operations are not feasible in React Native.

5. Evaluation

In this section, we will examine how well a mobile app performs when developed using two cross-platform development environments without causing any performance issues.   In Section 5.1, we first evaluate how well our integrated methodology fulfills various functional requirements by extensively experimenting a test scenario, which covers all tasks. In Section 5.2, we further investigate whether the transmission delay of a newly captured chroma key filtered image from a mobile app to a server causes any noticeable effect against users' experience after the image composition task.

For the experiment, we developed a mobile storybook app, using Unity3D (version 2019.2.17.f1) and React Native (version 0.62). All the cross-platform solutions produce binaries runnable on Android and iOS. Since our research group is more familiar with the Android development environment and has a variety of Android testing devices, we mainly conducted the testing on Android.

5.1 Reliability of Integration Methodology

This evaluation metric is intended to validate the reliability of our proposed integration methodology by observing abnormal behaviors or unexpected errors when performing complex image substitution task on various Android mobile devices. The image substitution task includes the all following actions in sequence.

1) Launch a mobile app.
2) Select a storybook from the list of storybooks randomly.
3) Select a page from the book randomly.
4) Select one of interactive images in the page randomly to replace the image.
5) Apply different background color for chroma keying and see whether an object of interest is extracted in the camera preview mode.
6) Take a photo from the chroma-key filtered camera preview mode.
7) Erase unwanted area from the photo.
8) Click to submit a button to send the chroma-key filtered image to the server.
9) Make sure newly changed image is placed on the right location.

Table 1. Eleven different types of Android devices and versions used for evaluation

Android Version	Model Name
10.0	Samsung Galaxy S10e, Samsung Galaxy S20, Samsung Galaxy Note 10+
9.0	LG V20, Samsung Galaxy Note 9
7.0	Samsung A8, Samsung A7
6.0	Samsung Galaxy S5, Samsung Note 4, Samsung A5
5.0	Samsung Galaxy Note 3

In the experiment, a total of 5 storybooks were preloaded on all Android devices, whose specifications are detailed in **Table 1**. During a preliminary experimentation before the evaluation, we found that there were small functional glitches on different Android version. On the LG V20, one of low-budget Android devices, running on Android version 8.0, the test app immediately crashed soon after the loading. This symptom was due to a minor design flaw by Android, which causes the app crash right after showing a splash screen (with full screen mode) starts when the app starts. We quickly fixed the problem by updating the Android version to 9.0. That's why Android version 8.0 was omitted in our testing environment shown

in **Table 1**. After fixing above issue, we also observed that one A7 device from Samsung, reported a strange behavior occurring when the chroma key shader didn't work properly in the graphics hardware of the device. In the meanwhile, there reported no problems on other devices during real-time chroma key filtering process. The problem only occurred when editing a photo taken during the photo-taking task. This problem was quickly corrected by using a software-based correction method, but its chroma key editing speed was slow compared to other devices.

For the verification, a total of 5 external users, who never used the mobile app before, participated in the experiment and were randomly assigned to fulfill the substitution task mentioned above using some of 11 mobile devices. In each storybook, the users were asked to select 5 pages randomly, execute image composition task on a randomly chosen image object on each page, and repeat the tasks three times. Thus, a total of 825 image composition tasks - that is, 5 (storybooks)×11 (mobile devices)×5 (pages)×3(repetition) - were performed. During the experiments, any abnormal operations such app crashes or freezing were considered failures regardless of their causes, such as network failure or server crashes.

During the experiments, we occasionally experienced transmission delays due to unexpected network transmission degradation at the server, but all 825 tasks were successfully executed. From this result, we confirmed that our integration methodology can operate with no significant performance issues even when experimenting complex image composition tasks.
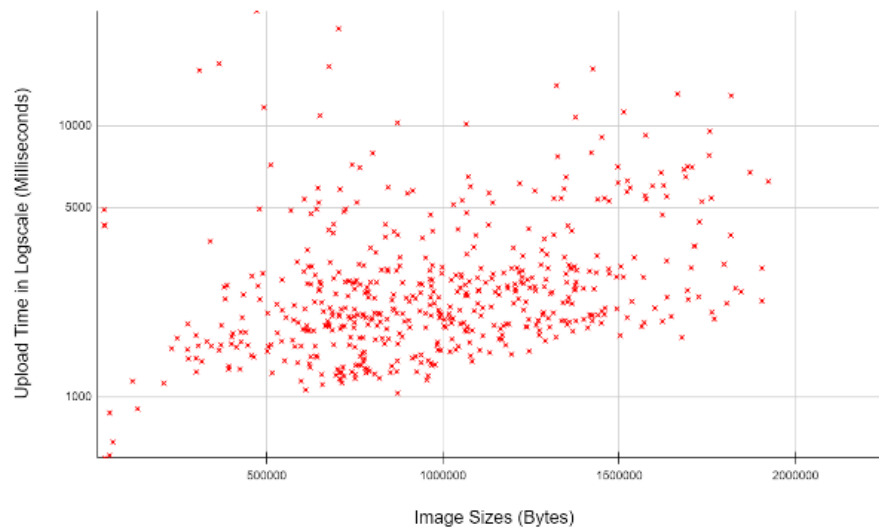

Fig. 8. Transmission Delays of Chroma-key Filtered Images from A Mobile App to a Server.

5.2 Effect Reliability of Integration Methodology

The data upload speed of chroma-keyed images depends on the image size. For a general compressed image file that is not chroma-keyed, the size greatly depends on the shooting resolution, typically averaging 2MB for a 1080×1920 image resolution. Therefore, reducing the image file size through chroma key processing has a significant impact on the response time performance.

During our experiments, we recorded the end-to-end transmission delays for all the images generated by chroma key processing. We then plotted the individual delays as a function of input chroma key filtered image sizes in **Fig. 8**. The figure shows the distribution of the upload response time (in milliseconds) of the image file extracted through the additional processing of the chroma key image by size (bytes) of the chroma key image file. The average response time is 2.962 seconds (±2.566) and the average image file size is 979KB (±391KB). The response time of the median value with reduced sensitivity according to the outlier value was 2.214 seconds, and the median file size was 954KB. This result indicates that the effect of any outliers was insignificant.

As shown in the figure, there was a rare spike in transmission delays during the test, but it was impaired soon after. For smaller image sizes, transfer rate is lower than for large images. The larger the image file is, the higher the transfer speed, but the greater than transfer variability. As shown in the figure, the transmission of a captured image after chroma keying to a server takes more than 2 seconds with no further post-processing at the server-side. Although an upload delay of around 2 seconds may be acceptable for many users, it still needs room for improvements. We expect that lowering filtered image size will reduce the transmission delays, improving the user experience for the app.

6. Conclusions and Future Work

Cross-platform technology allows developers to create software that can be used on multiple platforms with a single source code. This results in lower development and maintenance costs and higher productivity compared to native development methodologies. React Native is one of popularly used cross-platform development solutions that makes it easy to develop UI/UX related software components using proven mature web technologies. However, it is not suitable for working with pixel-level image manipulations in real time. On the other hand, Unity3D is equipped with libraries that allows developers to easily develop components that require complex calculations such as real-time image rendering, pixel-level image manipulation, and GPU-driven shader-heavy operations. But it is relatively slower when developing complex UI/UX—specific tasks.

In this study, we introduced a case study of developing a storybook mobile app with various functionalities, including storybook playback, editing, and image extraction. These functionalities are relatively complex and challenging to implement using a single cross-platform development tool. Instead of relying solely on one development environment, we propose a new methodology that combines different cross-platform development environments. We have verified through objective evaluations, conducted by external parties, that such integration can seamlessly integrate complex tasks. However, integrating more than two cross-platform solutions may not be appealing, as it could lead to concerns regarding developer availability, particularly for small-sized mobile app development companies. By leveraging well-recognized cross-platform solutions, we demonstrated that integration can be achieved in the development of complex tasks, as showcased in our customized storybook mobile app. In summary, we have proposed a new development method that integrates different cross-platform solutions when building highly complex and performant mobile apps. We have verified the feasibility of our method by developing a storybook mobile app.

Nevertheless, the integration process comes with certain consideration. While it is beneficial to leverage the advantages of individual cross-platform solutions, there may be unforeseen issues such as sandbox redundancy or unexpected functional burdens, including redundant permission requests. Despite these challenges, we believe that combining two cross-

platform solutions can be applicable to various functional and performance requirements, enabling the development of more sophisticated mobile applications at reduced costs and shorter development timelines.

Despite meeting all functional requirements, our mobile app's user experience could be improved. To maintain high-quality object extraction, we combined chroma-key filtering and manual editing, which results in relatively high transmission delays (over 2 seconds on average in our tests) when uploading chroma-keyed image files, which degrades the user experience. Therefore, reducing image files while maintaining extraction quality would significantly improve the user experience.

The latest deep learning-based image segmentation methods [25] can extract desired objects more clearly and separating chroma-key masks from images, enabling independent delivery of resulting image files, is expected to further reduce transmission delay.

References

[1]   Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli, "Progressive Web Apps: The Possible Web-native Unifier for Mobile Development," in *Proc. of the 13th International Conference on Web Information Systems and Technologies WEBIST*, pp. 344-351, 2017. Article (CrossRef Link)

[2]   Tina Beranic, Patrik Rek, and Marjan Hericko, "Adoption and Usability of Low-code/no-code Development Tools," in *Proc. of Central European Conference on Information and Intelligent Systems*, pp. 97-103, 2020.

[3]   JetBrains, "Cross-platform Mobile Frameworks used by Software Developers Worldwide from 2019 to 2021," *Statista*, 2021. [Online] Available: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours

[4]   Gabriel Peal, "Sunsetting React Native," *Airbnb Tech Blog (Medium)*, 2018. [Online] Available: https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a

[5]   Andreas Biørn-Hansen, Tor-Morten and Grønli and Gheorgphita Ghinear, "A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development," *ACM. Computing Surveys*, vol. 51, no. 5, pp. 1-34, 2018. Article (CrossRef Link)

[6]   Timothy Yudi Adinugroho, Reina, and Josef Bernadi Gautama. "Review of multi-platform mobile application development using webview: Learning management system on mobile platform," *Procedia Computer Science*, vol. 59, pp. 291-297, 2015. Article (CrossRef Link)

[7]   Vipul Kaushik, Kamali Gupta, and Deepali Gupta, "React Native Application Development," *International Journal of Advanced Studies of Scientific Research*, vol. 4, No. 1, 2019. Article (CrossRef Link)

[8]   Larry Heimann, and Oscar Veliz, "Mobile Application Development in Flutter," in *Proc. of the 53rd ACM Technical Symposium on Computer Science Education V.2*, pp. 1199-1199, 2022. Article (CrossRef Link)

[9]   Dan Hermes, *Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals*, 1st ed. Berkeley, CA, USA: Apress, 2015. Article (CrossRef Link)

[10]   Jingming Xie, "Research on Key Technologies based Unity3D Game Engine," in *Proc. of 2012 7th International Conf. on Computer Science and Education (ICCSE)*, pp. 695-699, 2012. Article (CrossRef Link)

[11]   Andrew Sanders, *An introduction to Unreal engine 4*, A. K. Peters, Ltd., 2016. Article (CrossRef Link)

[12]   A. Smith, and J. Blinn, "Blue Screen Matting," in *Proc. of SIGGRAPH '96*, pp. 259-268, 1996. Article (CrossRef Link)

[13]   Swarnendu Ghosh, Nibaran Das, Ishita Das, and Ujjwal Maulik, "Understanding Deep Learning Techniques for Image Segmentation," *ACM Computing Surveys*, vol. 52, no. 4, 2019, Article no. 73. Article (CrossRef Link)

[14] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for MobileNetV3," in *Proc. of 2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1314-1324, 2019. Article (CrossRef Link)

[15] Machine Learning Research at Apple, "On-device Panoptic Segmentation for Camera using Transformers," Oct. 2021. [Online] Available: https://machinelearning.apple.com/research/panoptic-segmentation

[16] Barbara Leporini, and Clara Meattini, "Personalization in the Interactive EPUB 3 Reading Experience: Accessibility Issue for Screen Reader Users," in *Proc. of the 16th International Web for All Conference*, pp. 1-10, 2019. Article (CrossRef Link)

[17] R. Battle, and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)," *Journal of Web Semantics*, vol. 6, no. 1, pp. 61-69, 2008. Article (CrossRef Link)

[18] Francois Beaulieu, "Part 1. Show Unity3D view in React-Native application. Yes it's possible!," Feb. 2018. [Online] Available: https://medium.com/@beaulieufrancois/show-unity3d-view-in-react-native-application-yes-its-possible-852923389f2d

[19] React Native Unity View, "react-native-unity-view," Jan. 2019. [Online] Available: https://www.npmjs.com/package/ react-native-unity-view Accessed on: Feb 1, 2023

[20] Software Mansion, "About React Native Reanimated," [Online] Available: https://docs.swmansion.com/react-native-reanimated/docs/ Accessed on: Apr. 6, 2023

[21] T. Hidayat, and B. D. Sungkowo, "Comparison of Memory Consumptive Against the Use of Various Image Formats for App Onboarding Animation Assets on Android with Lottie JSON," in *Proc. of 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE)*, Yogyakarta, pp. 376-381, 2020. Article (CrossRef Link)

[22] Wenjie Zou, Jiarun Song, and Fuzheng Yang, "Perceived Image Quality on Mobile Phones with Different Screen Resolution," *Mobile Information Systems*, vol. 2016, 17 pages, 2016, article ID 9621925. Article (CrossRef Link)

[23] Rob Pike, Bart Locanthi, and J. Reiser, "Hardware/software Trade-offs for Bitmap Graphics on the Blit," *Software: Pactice and Experience*, vol. 15, no. 2, pp. 131-151, 1985. Article (CrossRef Link)

[24] Chroma key shader asset for Unity, "uChromaKey," 2021. [Online] Available: https://github.com/hecomi/uChromaKey

[25] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Tete, S. Whitehead, A.C. Berg, W.-W. Lo, P. Dollar, R. Girshick, "Segment Anything," in *Proc. of the IEEE/CVF Internaltional Conf. on Computer Vision*, pp. 4015-4026, Oct. 2023. Article (CrossRef Link)

Beomjoo Seo, He received the B.S. and M.S. degrees from the Department of Computer Engineering, Seoul National University in 1994 and 1996, respectively, and the Ph.D. degree in Computer Science from the University of Southern California in 2008. He was formerly a Senior Research Fellow at the School of Computing, National University of Singapore. He is currently an assistant professor at the School of Games in Hongik University.