

# P2A: A Tool for Converting Pixels to Animated Mobile Application User Interfaces

Siva Natarajan, Christoph Csallner  
Computer Science and Engineering Department  
The University of Texas at Arlington  
Arlington, TX, USA  
siva.natarajan@mavs.uta.edu, csallner@uta.edu

## ABSTRACT

Developing mobile applications is typically a labor-intensive process in which software engineers manually re-implement in code screen designs, inter-screen transitions, and in-screen animations developed by user interface and user experience experts. Other engineering domains have used computer vision techniques to automate human perception and manual data entry tasks. The P2A tool adopts computer vision techniques for developing animated mobile applications. P2A infers from mobile application screen designs the user interface portion of an application's source code and other assets that are ready to be compiled and executed on a mobile phone. Among others, inferred mobile applications contain inter-screen transitions and in-screen animations. In our experiments on screenshots of 30 highly-ranked third-party Android applications, the P2A-generated application user interfaces exhibited high pixel-to-pixel similarity with their input screenshots. P2A took an average of 26 seconds to infer in-screen animations.

## CCS CONCEPTS

• **Human-centered computing** → User interface programming; Ubiquitous and mobile computing design and evaluation methods; • **Software and its engineering** → Integrated and visual development environments; Source code generation;

## KEYWORDS

Mobile software engineering, pixel-based design

### ACM Reference Format:

Siva Natarajan, Christoph Csallner. 2018. P2A: A Tool for Converting Pixels to Animated Mobile Application User Interfaces. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, May 27–28, 2018, Gothenburg, Sweden. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3197231.3197249>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
*MOBILESoft '18*, May 27–28, 2018, Gothenburg, Sweden  
© 2018 Association for Computing Machinery.  
ACM ISBN 978-1-4503-5712-8/18/05...\$15.00  
<https://doi.org/10.1145/3197231.3197249>

## 1 INTRODUCTION

Developing the user interface portion of a successful mobile application currently requires mastery of several disciplines, including graphic design and programming. A key reason is that consumers have come to expect screen designs, inter-screen transitions [54], and in-screen animations that are innovative and highly customized [38, 41]. While many non-programmers are well equipped to design such a user interface, these people currently have to collaborate with software engineers to deliver a working mobile application user interface.

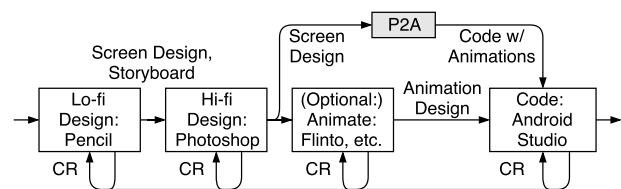


Figure 1: Excerpt of a typical mobile application development process (bottom); each box is an activity; each transition arrow is annotated with artifacts; CR = change request. Adding P2A automates the current manual step of recreating screen and animation designs in code.

Concretely, the bottom of Figure 1 shows a typical app development process. User experience (UX) and user interface (UI) experts go through several iterations of designing storyboards as well as low-fidelity and high-fidelity screen designs, using a sequence of customary tools such as paper-and-pencil and Photoshop [40]. In addition to these designs, UX experts develop detailed descriptions of the in-screen animations and inter-screen transitions. To capture such descriptions, some teams use a dedicated tool such as Adobe Flash, Flinto, FramerJS, or Invision [11, 17, 26, 38] to produce throw-away animation prototypes. Such prototypes require emulators and often fail to emulate animation and transition effects faithfully.

The next step is a gap in the production pipeline [22], as the produced artifacts have to be handed to a different kind of people, software engineers who are familiar with the target mobile platform's coding languages, e.g., XML and Java for Android. The engineers then manually recreate the screen and animation designs in code, which is cumbersome, error-prone, and expensive. Unwelcome surprises may wait at the end of this process, as teams discover that the native application animation and transition effects differ from the throw-away animation prototypes.

The production gap, essentially between non-programmers and programmers, is a hard problem in practice. Programming effectively has a high barrier to entry, which makes it difficult to recruit, train, and retain non-programmers. This creates a bottleneck on the number of programmers and artificially restricts non-programmers' ability of producing working application user interfaces.

However the popularity of mobile devices has created a large demand for mobile applications. For example, In the USA over 90% of consumers over 16 years of age use a mobile phone and more than half of the mobile phones are smartphones, mostly running Android or iOS [48]. On these smartphones, people use mobile applications to perform many tasks that have traditionally been performed on desktop computers [3, 15, 25, 48]. Example tasks include reading and writing emails, listening to music, watching movies, reading the news, and consuming and producing social media. To date, more than one million mobile applications have been released<sup>1</sup>. Given their wide use, it would be beneficial to empower more people to create larger portions of such applications.

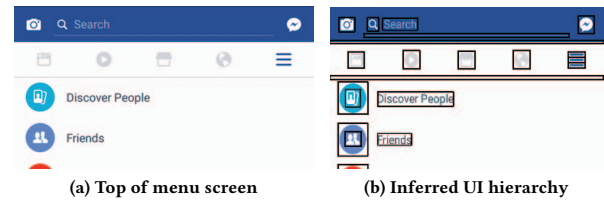
Converting screen designs into good user interface code is hard, as it is essentially a reverse engineering task—general principles have to be inferred from specific instances [37]. For example, a suitable hierarchy of user interface elements has to be inferred from a flat set of concrete pixels. Compared to other reverse engineering tasks, an unusual additional challenge is that the input, i.e., the pixels, may originate from scanned pencil sketches with all their imperfections [4, 39, 49]. This means that sets of pixels have to be grouped together and recognized heuristically as images or text. Then groups of similar images and text have to be recognized heuristically as example elements of collections. And for the user interface of innovative mobile applications, at each step the recognized elements may diverge significantly from the platform's standard user interface elements.

For professional application development, one may wonder if this reverse engineering step is artificial. That is, why are meaning and source code hierarchy of screen elements not explicitly encoded in the screen designs if these are done in digital tools such as Photoshop? One reason is that some UX designers start with pencil on paper, so it would be desirable to convert such drawings directly into working user interface code. More significantly, when UX designers create digital bitmap images (typically by drawing them in Photoshop), the digital design tools do not capture the hierarchy information that is needed by user interface code. More importantly, it is not clear if UX designers and graphic artists want to think in terms of source code hierarchies.

The most closely related work is REMAUI [37]. Similar to P2A, REMAUI takes as input a screen design in pixel form and converts it to suitable UI code. However REMAUI is limited to processing one input screen bitmap per app and does not support inter-screen transitions or in-screen animations.

This paper addresses the problem of inferring animated application user interfaces directly from screen designs. We can use computer vision techniques to reverse engineer UI code hierarchies from pixels. Then we expose these reverse engineered UI view hierarchies to the user, to enable the user to define inter-screen transitions interactively within screen designs. Specifically, P2A

<sup>1</sup><http://www.appbrain.com/stats/number-of-android-apps>



**Figure 2: Original Facebook Android app menu screen (left) and the same screen super-imposed with P2A-inferred UI elements (right).**

first uses REMAUI to identify different visual elements, arrange them in a hierarchy, and superimpose this hierarchy on the input screen designs. The key insight of this paper is that from similar UI elements on multiple input screens we can also infer in-screen animations, even if similar UI elements differ significantly across screens. We can then merge the involved screens into a single animated screen, and export the results as native mobile application source code and other assets that are ready to be compiled and executed on stock mobile phones.

In our experiments on screenshots of 30 highly-ranked third-party Android applications, the P2A-generated application user interfaces exhibited high pixel-to-pixel similarity with their input screenshots. P2A took an average of 26 seconds to identify in-screen animations. To summarize, the paper makes the following major contributions.

- The paper introduces the first technique to automatically infer from pixel-based screen designs the user interface portion of animated applications.
- The paper evaluates this technique by implementing a prototype for Android and running the prototype on 30 highly-ranked third-party Android applications.

## 2 MOTIVATING EXAMPLE: FACEBOOK APP

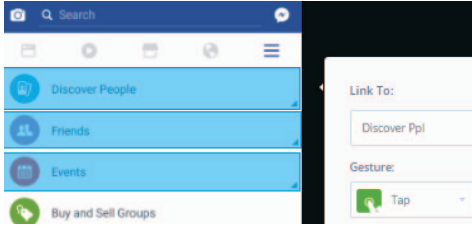
As a concrete motivating example, consider an UI designer creating an interactive prototype for the recent version 105.0.0.23.137 of the Facebook Android app.

### 2.1 Inter-screen Transitions

Figure 2a shows the top of the Facebook Android app menu screen. For example, tapping the text “Discover People” should trigger a transition to another screen via a sliding animation.

State-of-the-art app animation prototyping tools such as Invision and Flinto remain at the pixel level. They take as input pixel-based screen designs (e.g., the Figure 2a screen) and allow the user to interactively mark image areas of interest as “hotspots” at the pixel level [18, 21]. For example, Figure 3 shows the Facebook menu screen in Invision, where the user has drawn three hotspot areas and is configuring one hotspot’s interaction behavior.

In contrast, P2A infers from the input image the code structures the target mobile app could use to display a screen similar to the input image. P2A visualizes such inferred UI elements and lets the user interactively select them. For example, Figure 2b shows the UI structure P2A inferred for the Figure 2a input image. P2A can



**Figure 3: An Invision user marks up the screen at the pixel level (the large rectangles) and specifies screen transitions, e.g., from the menu screen to the “Discover People” screen.**

thus readily generate corresponding source code for the target platform, complete with user-specified screen transitions. The resulting P2A-generated animation looks like the original animation in the Android Facebook app.

To get an idea of the overall workflow of representative state-of-the-art tools, we conducted a small exploratory case study of creating a basic interactive Android prototype with 10 screens and a total of 10 corresponding inter-screen transitions. Table 1 summarizes the tools’ runtime on recent versions of Invision and Flinto on a standard desktop computer.

Task	Invision	Flinto	P2A
Load & preprocess screens	38	12	103
Assign inter-screen transitions	122	192	135
Generate target platform code	n/a	n/a	2
Total [s]	160	204	240

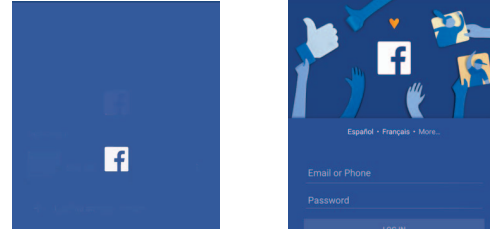
**Table 1: Example runtime for creating a prototype with 10 inter-screen transitions. Invision and Flinto prototypes only run in an emulator. P2A creates an Android app.**

Invision and Flinto took a few seconds to load and preprocess the input screen designs, with the bulk of the runtime going to the user interactively defining inter-screen transitions. For both tools this yielded a throw-away prototype that a developer then has to manually re-create in XML and Java, which typically requires several hours. In contrast, a P2A user required under 4 minutes to process the images and define inter-screen transitions. Within this time P2A also generated all source code and assets ready to be compiled for stock Android devices.

## 2.2 In-screen Animations

As an example in-screen animation, consider the Facebook Android app’s login screen animation from Figure 4a to 4b, where the Facebook logo at the same time changes its color, slides upwards (“translation”), and expands (“scaling”). After these changes the other elements of Figure 4b fade in together. The evaluation contains cases in which animated screen elements differ more significantly than the Figure 4 Facebook “f” and our prototype still matches them and infers correct animations.

Similar to defining inter-screen transitions, the most closely related approaches take as input pixel-based screen designs, require



(a) Initial state

(b) Final state

**Figure 4: Facebook Android app login screen animation.**

the user to define animations manually, yield emulator-based prototypes, and do not produce any code for the target platform. In contrast, P2A infers animations from the input screens and emits native UI application code ready for compilation and execution.

Given two related input screens, P2A identifies shared elements via perceptual hashing and determines how to transition one shared element instance into the other. P2A removes or adds non-shared elements after the animation end. For example, for the Figure 4a and 4b start and end screens, P2A emits the Listing 1 Java code. Specifically, the emitted code translates the logo upward (Lines 2 & 3 define the logo’s new location) and scales it (Lines 4 & 5 define its new dimensions).

```
public void onStartAnimation()
{ // ..
  f = (RelativeLayout.LayoutParams) v.getLayoutParams();
  f.leftMargin -= 11; // (2)
  f.topMargin -= 324; // (3)
  f.width = 56; // (4)
  f.height = 58; // (5)
  // ..
}
```

**Listing 1: Condensed excerpt of the in-screen animation code P2A inferred from the Figure 4 input images, which faithfully reproduces the Android Facebook app animation.**

To get an overview of the overall workflow of the most closely related approaches, we performed a small exploratory case study creating this animation on a standard desktop computer. Specifically, creating the Facebook login animation using either Adobe Flash or FramerJS took about 15 minutes. The resulting prototypes require an emulator and manual translation to code for the target platform. In contrast, P2A took under 30 seconds to automatically detect the animation and create XML and Java code that is ready to be compiled and produces the intended animation on a stock Android device.

## 3 BACKGROUND

This section contains necessary background information on graphical user interfaces, the example Android GUI framework, and perceptual hashing.

### 3.1 Graphical User Interface (GUI)

The graphical user interface (GUI) of many modern desktop and mobile platforms is structured as a view hierarchy [1, 33]. Such

a hierarchy consists of two types of nodes, leaf nodes (images, buttons, text, etc.) and container nodes. The root view represents an application's entire space on screen. The root can have many transitive children. Each child typically occupies a rectangular sub-region of its parent. Each view can have its own parameters such as height, width, background color, and position.

A mobile application typically consists of several distinct screens (e.g., a list overview screen and a list item detail screen). Each screen has its own view hierarchy. The application transitions between screens, typically triggered by user input. The GUI framework receives user input via the OS from the hardware and passes them as events to the view hierarchy node the user interacted with (e.g., a screen touch event to the leaf node that occupies the screen position the user touched). A screen can be animated. An in-screen animation may maintain or manipulate the existing view hierarchy.

To define basic GUI aspects, modern platforms provide two alternatives. The traditional desktop approach is construction through regular program code [33]. The now widely recommended alternative is declarative [1, 16, 35], e.g., via XML layout definition files in Android. Advanced GUI aspects such as inter-screen transitions and in-screen animations are then defined programmatically, which typically leads to a combination of code and layout declaration files.

Building an appealing user interface is hard [32, 33]. Besides understanding user needs, standard GUI frameworks are complex and offer many similar overlapping concepts. This challenge is especially significant for developers new to their target platform. While each platform provides standard documentation and sample code, these samples often produce unappealing results.

### 3.2 Example: GUI Framework of Android

At a high level, an Android app is defined as a set of activities. Each activity creates a window that typically fills the entire screen. While Android also has smaller non-fullscreen activities such as menus and pop-up window style dialogs, this paper focuses on fullscreen activities, since fullscreen activities are typically more complex with deeper view hierarchies.

Android applications typically contain more than one activity [2, 44, 53]. For example, according to an August 2012 survey of the 400 most popular non-game applications in the Google Play app store [44], the average number of total activities per app ranged from 1.3 (in "Software & Demos") to 61.1 ("Finances"). A set of widely used Android benchmark applications ranges from 2 to 50 screens (plus 2 to 20 menus and zero to 48 dialogs), defined by 53 to 1,228 classes with a total of 241 to 5,782 methods [53].

The Android standard libraries define various containers ("view groups") and leaf nodes ("widgets"). According to the August 2012 survey [44], the following containers were used most frequently: `LinearLayout` (130 uses per application on average) places its children in a single row or column; `RelativeLayout` (47) positions children relative to itself or each other; `FrameLayout` (15) typically has a single child; `ScrollView` (9) is a scrollable `FrameLayout`; and `ListView` (7) lays out children as a vertical scrollable list.

The following widgets were used most frequently: `TextView` (141) is read-only text; `ImageView` (62) is a bitmap; `Button` (37) is a device-specific text button; and `View` (17) is a generic view. Besides the

above, the Android library documentation currently lists dozens additional standard widgets and containers.

The following are key Android event types for interacting with views: click is a sequence of press down, no moving, and release—typically on a touch screen; swipe is a press down, move, release sequence; long click is a click that exceeds a system-defined duration; and item click is a click on an item, typically in a list or other collection. Before acting on an event, the Java method ("callback") defining the action must be bound ("registered") to a view and an event type either in XML or in Java code.

An inter-screen transition ("activity transition") is typically triggered by a user action such as a click on a certain view. Since Android 5.0 (API Level 21) and its Material Design [19], Android supports transition effects. Each inter-screen transition can define how views should exit the source activity and enter the target activity. The following are example predefined Android transition effects: slide is a move through a window edge; explode is a move through the window center; and fade is a change of opacity.

An in-screen animation ("transition") may manipulate an activity's view hierarchy and specify how views are added or removed. During a transition, each view may also be animated ("property animation"). Following are example animations: translate is a change of on-screen position; scale is a change of dimensions; rotate is a rotation; and alpha controls opacity.

### 3.3 Perceptual Hashing

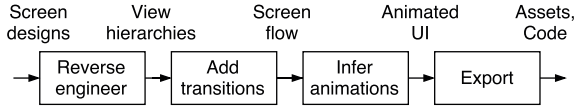
Perceptual hashing takes as input an image and produces a "fingerprint", i.e., a fixed-length hash value [30, 31, 46, 55]. The key motivation of perceptual hashing is that if two images look similar to a human observer, then their perceptual hash values should also be similar. This is a common goal, e.g., for finding similar media files on the web. For example, for a given image its copyright holder may want to find all variants online, even if they have different watermarks or contain other small variations such as being darker, cropped, or compressed.

At a high level, perceptual hashing algorithms perform strong compression, drastically reducing the image size, while identifying and encoding an input image's essential properties. These properties are unlikely to change during image modifications that yield variants that appear similar to a human observer.

## 4 OVERVIEW AND DESIGN

At a high level, the P2A workflow (Figure 5) consists of the following four major phases. (1) First, P2A uses REMAUI to reverse engineer each pixel-based input screen into a corresponding UI view hierarchy. P2A displays this hierarchy as a transparent overlay over the input screen, e.g., as in Figure 2b. (2) Second, to specify an inter-screen transition, the user selects a screen overlay's UI hierarchy element and interactively connects it with a target screen.

(3) Third, in screen pairs that may represent the start and end state of an in-screen animation, P2A identifies matching view elements and infers corresponding in-screen animations, thereby merging two (non-animated) input screens into one (animated) screen. (4) Finally, P2A infers UI code that produces screens similar to the input screens, inter-screen transitions, and in-screen animations as assets and source files that are ready to be compiled and



**Figure 5: Flow diagram of P2A’s four major phases. Phase transitions are annotated with the inferred artifacts.**

executed on the target mobile platform. Following are key elements of these four phases.

#### 4.1 Input Screens to UI View Hierarchies

P2A takes as input a set of pixel-based screen designs, where each screen is given as a image bitmap. P2A first uses REMAUI to process each input screen in isolation. In each input bitmap, P2A identifies user interface elements such as images, text, containers, and lists, via a combination of computer vision and optical character recognition (OCR) techniques. To detect mobile app UI elements, P2A follows the REMAUI approach [37] and combines off-the-shelf computer vision, OCR, and domain-specific heuristics. P2A further infers a suitable user interface hierarchy, one hierarchy per input screen design and visualizes each inferred UI hierarchy as an overlay over the input screen design.

At the high level, this step uses a sequence of standard computer vision techniques for edge detection and determining bounding boxes around groups of pixels that likely belong together from a user perspective, i.e., Canny’s algorithm [7, 47], followed by (edge) dilate, and contour. P2A uses domain-specific heuristics to combine pixel regions that are close to each other and likely belong to the same image or word. P2A further combines computer vision, OCR, and heuristics to distinguish images from text. Such image detection and OCR tasks are heuristic in nature, but the underlying REMAUI approach has been reasonably accurate for the mobile screen domain [37].

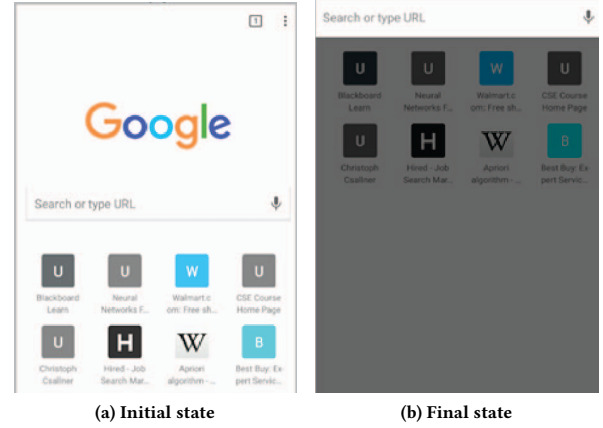
P2A extracts OCR-inferred text strings. P2A also identifies collections, by comparing close-by view hierarchy subtrees. Finally, P2A exposes the inferred UI element bounding boxes to the user, by superimposing them on the input screen designs.

#### 4.2 Adding Inter-Screen Transitions

P2A allows the users to interact with the inferred UI hierarchies and create inter-screen transitions complete with transition effects. Specifically, the user can select an individual inferred bounding box and interactively connect it to a target screen.

To customize the specified inter-screen transition, the user can select the type of event that should trigger the transition. P2A currently supports the common event types click, long click, and item click. To further customize an inter-screen transition, the user can select transition effects. P2A currently supports the common slide, fade, explode, and shared element effects.

In practice, interacting with a single UI element may trigger different inter-screen transitions. For example, clicking and long-clicking on a screen element may transition an app to different screens. To support such multiple transitions per screen, P2A users can specify a transition for each element/event pair.



**Figure 6: Android Chrome application landing screen states. Screen elements are similar. But since they differ in brightness they differ widely according to common pixel-by-pixel comparison metrics such as MSE.**

#### 4.3 Inferring In-screen Animations

Given a screen pair, P2A at a high level emulates how a human user may infer in-screen animations between the two given screens, using the following three-step process. (1) First, P2A efficiently matches elements that are similar on both screens except for their location, size, and color. (2) Second, P2A removes from animation consideration elements that change very little or not at all between the two screens. (3) Finally, P2A merges the two input screens and creates in-screen animation code that animates the transition from start to final animation state. Following are these steps’ key elements.

**4.3.1 Matching (Non-Text) Images Across Screens.** At this point, P2A has already broken each input screen image down into elements (via identifying bounding boxes, etc.). So conceptually we can now iterate over the screen elements and match them with corresponding elements of the other screen. To compare two images, for decades computer vision research and practice have relied heavily on pixel-by-pixel comparisons, e.g., by using the Mean Squared Error (MSE) metric [50].

However this comparison of screen elements is tricky, as a basic pixel-by-pixel comparison will not work in many cases, e.g., for matching an element E to a zoomed version of E, e.g., the resized Facebook logo in the login screen animation of Figures 4a and 4b.

More significantly, another common scenario in which pixel-by-pixel comparisons do not work well is if an input screen is an exact copy of the other except that elements in one screen differ in color or brightness, e.g., because they are “grayed out”. Figure 6 is an example. Both screens share several similar elements. However a pixel-by-pixel comparison metric would still tell us that the elements differ widely, as the right screen hides the screen elements behind a semi-transparent layer.

To address such problems with pixel-by-pixel comparisons, more advanced comparison metrics have been developed such as SSIM,



which take into account each pixel's immediate surrounding pixels [50]. While SSIM addresses our color and brightness (e.g., Figure 6) problems and we could likely overcome the scaling (e.g., Figure 4) problem by scaling both images to the same size, SSIM is still fundamentally a technique for comparing a pair of images. Assuming we have identified  $e$  elements in one screen,  $f$  elements in the other screen, and that during comparisons (after scaling) a screen element has  $p$  pixels, performing pixel-by-pixel comparisons or SSIM just on two given screens has complexity  $O(e * f * p)$ .

To overcome the comparison quality and runtime problems, P2A uses perceptual hashing, which is more accurate in identifying the similarity between images under modifications such as scaling or changes in color intensities.




Our perceptual hashing algorithm is relatively straightforward [55]. The first step is resizing the input image to a common resolution (e.g., 32x32). On the resized image we apply the Discrete Cosine Transformation (DCT). We then select the higher significant components from the top-left 8x8 and discard the rest of the image. Finally, we generate a 64-bit hash value using the reduced DCT components (8x8).

The complexity of computing the perceptual hash of one screen element is  $O(p)$  and for all elements of two screens  $O(p * (e + f))$ . We reduce the overall runtime complexity by upfront computing and caching the perceptual hash of each P2A-identified element.

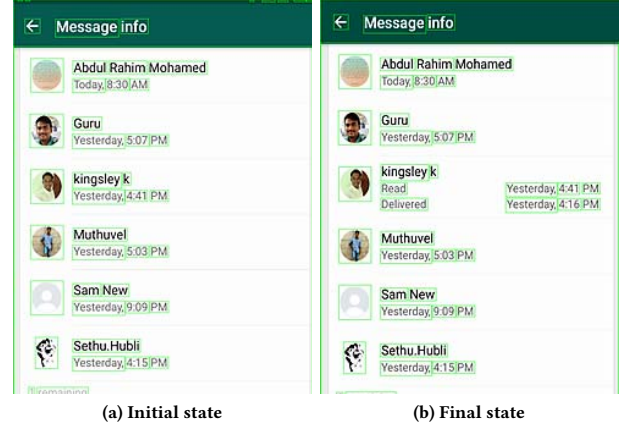
We then compare the resulting perceptual hashes efficiently, by computing their Hamming distance. Recall that the Hamming distance between two strings of equal length is the number positions at which the corresponding symbols differ. For example, the Hamming distance between 1011101 and 1001001 is two. If two images appear similar to a human observer, the Hamming distance between the images perceptual hash values is small. Two identical images yield the same perceptual hash value, which yields a Hamming distance of zero.

In our implementation, which is similar to pHash [55], each resulting hash value is a 64 bit binary. We can thus compute the Hamming distance of two hash values efficiently in constant time, typically via the two hardware instructions xor and popcount (where the latter counts the number of ones). The complexity for computing the Hamming distance between all pairs of screen elements is thus  $O(e * f)$ . But since we typically only have a few dozen screen elements ( $e \ll p$ ), the overall complexity remains  $O(p * (e + f))$ .

Based on our experiments on half of the input images, P2A uses a Hamming distance heuristic threshold of 5. Specifically, P2A considers two screen elements similar if the Hamming distance between their perceptual hash values is less than 5.

Image			
pHash	104efec61bdd9	304ffec61bff9	1fdfff3efe47f
H-dist	0	4	24

**Table 2: Perceptual hash (pHash) values of three example images in hex and their Hamming distance from the left "f" logo (104efec61bdd9).**



**Figure 7: Example WhatsApp Android screens.**

As a concrete example, Table 2 compares the perceptual hash values of three example P2A-inferred screen elements. Since the two version of the Facebook logo are similar to one another, the Hamming distance between their perceptual hashes (in binary) is 4, less than our threshold. For the left and right images, the Hamming distance between their perceptual hashes is 24, greater than our threshold, as expected due to their large visual difference.

**4.3.2 Matching Text & Ignoring Minimal Changes.** At this point we have already run OCR over the input images and identified text boxes and their text content strings via a combination of computer vision, OCR, and domain-specific heuristics. We thus use the extracted text strings to efficiently match text boxes across screens.

Using the Figure 7 example, it is clear that in two screens a lot of elements may match. Many of these may be at slightly different positions or slightly different scales, due to differences in how the two input screens were designed. However such small differences should typically be ignored. So the algorithm uses a small customizable heuristic threshold of a 10-pixel position or dimension difference ( $\Delta_{pos} < 10$ ) & ( $\Delta_{dim} < 10$ ) to prune screen elements that have similar versions in both screens at very similar locations.

For example, the two Android WhatsApp version 2.17.107 screens of Figures 7a and 7b contain many "Yesterday" strings. Initially these "Yesterday" text elements all match with each other, yielding many possible animations. However using our 10-pixel heuristic we prune elements that have a match at almost the same location in both screens. This quickly reduces the number of matches and thus the number of animations, leaving us only with matching elements that are different enough to produce an animation that is clear enough to be reliably recognized by users.

#### 4.4 Inferred App Source Code & Assets

P2A generates the source code and asset files, compiles them, and bundles them together into an executable file. For Android, apart from generating compilation ready Java source code, P2A packages the generated source code and other assets such as layout.xml, strings.xml, and cropped screen element images into an Android executable (.apk) file.

Developers can consume the resulting artifacts in a typical mobile app development process. For example, the generated application contains one layout file for each input screen design and one Activity class for each input screen design. In the case of an in-screen animation the initial and final state are part of the same activity.

```
public class MenuAct extends Activity
implements View.OnClickListener { /* ... */
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.TextView_11:
                Intent in = new Intent(this, AmazonHomeAct.class);
                Slide anim = new Slide();
                getWindow().setEnterTransition(anim);
                getWindow().setExitTransition(anim);
                ActivityOptions ao = ActivityOptions.
                    makeSceneTransitionAnimation(this, null);
                break;
        }
    }
}
```

**Listing 2: Code snippet (condensed) of a generated Activity class that contains an user-defined inter-screen transition.**

Listing 2 shows a part of an example generated Android Java code snippet. Intent is the message passing mechanism in Android. The code instantiates the user-specified slide transition effect and the startActivity method navigates to a new screen.

## 5 RESEARCH QUESTIONS (RQ), EXPECTATIONS (E), AND HYPOTHESES (H)

To evaluate P2A, we ask (a) if it is currently feasible to integrate P2A into a standard mobile application development setup and (b) if P2A-generated user interfaces are useful in the sense that the generated UI is similar to the UI an expert developer would produce. We therefore investigate the following three research questions (RQ), expectations (E), and hypotheses (H).

- RQ1: What is P2A’s runtime for generating code for inter-screen transitions?
  - E1: Given P2A’s prototype status, we do not expect P2A’s runtime to allow interactive use.
  - H1: P2A can generate inter-screen transition code within a few seconds.
- RQ2: What is P2A’s runtime for inferring in-screen animations?
  - E1: Given the complexity of computer vision and OCR, we do not expect P2A’s runtime to allow interactive use.
  - H1: P2A can infer some in-screen animations within one minute.
- RQ3: Does a P2A-inferred in-screen animation look visually similar to the original animation?
  - E1: Given the complex nature of some in-screen animations, we do not expect P2A to achieve perfect similarity.
  - H1: P2A-inferred in-screen animations are visually similar to subjects’ original animations.

## 6 EVALUATION

To explore our research questions, we have implemented a prototype P2A tool that generates Android applications that are ready to

be compiled and executed on stock Android devices. Among others, the generated applications contain user-specified inter-screen transitions and P2A-inferred in-screen animations.

While P2A could attempt to search all input screens for matching elements for in-screen transitions, our current prototype focuses on inferring in-screen animations between a given animation start-state screen and a given animation end-state screen. P2A is implemented on top of REMAUI, and sequentially runs REMAUI once for each input screen design. Optimizing this process (e.g., to reuse intermediate results across input screens) is part of future work. For the experiments we used a 2.7Ghz Intel Core i5 Mac Book Pro with 16GB of RAM.

### 6.1 Subjects From Top-100 Free Android Apps

While the goal of P2A is to convert pixel-based screen designs to application code, in this paper we do not directly evaluate P2A on actual screen designs. The main reason is that it is hard to acquire a large number of third-party screen designs and compare P2A’s output with the screen designers’ intended application behavior. For this evaluation it was easier to leverage existing applications and use their screenshots as screen designs. In some sense this simulates an ideal scenario in which the developers are able to faithfully translate a UX designer’s screen designs to application behavior.

For this evaluation we have selected our subjects from the top-100 free apps of the Google Play store [20] in Apr 2017. From these we removed all games from consideration, as they typically use a different GUI paradigm such as OpenGL.

For the first experiment (RQ1) we focused on inter-screen transitions. Since not all top-100 apps contained transition effects, we selected those applications that contained more than 5 inter-screen transition effects. This left us with 10 applications comprising of 107 screens.

For our second experiment (RQ2 & RQ3) we focused on in-screen animations. We went through all screens of the top-100 apps in a depth-first manner in search of an in-screen animation. Only 30 applications contained an in-screen animation. In each app, for the first in-screen animation we encountered, we captured a screenshot before and after the animation. This yielded a dataset of 30 in-screen animations comprising of 60 screenshots.

The following roughly classifies the 30 in-screen animation subjects by the animation types they used for their elements. Specifically, 10 applications only used element fade-in, 2 only used translation, and 1 only used scaling. The remaining applications used a combination of animation types, i.e., scaling and translation (6 subjects), translation and fade-in (4), as well as fade-in and fade-out (3). Other animation combinations occurred only once, e.g., Walmart was the only one to use translation, fade-in, and scaling; Waze was the only one to use translation, fade-in, and fade-out.

### 6.2 RQ1: Runtime of Inter-Screen Transitions

Figure 8 shows the runtime of our 10 RQ1 subjects for a varying number of inter-screen transitions. Specifically, we used P2A to specify subsets of the subjects’ transitions and measured P2A’s corresponding runtime. As expected, tool runtime increased with the number of inter-screen transitions.

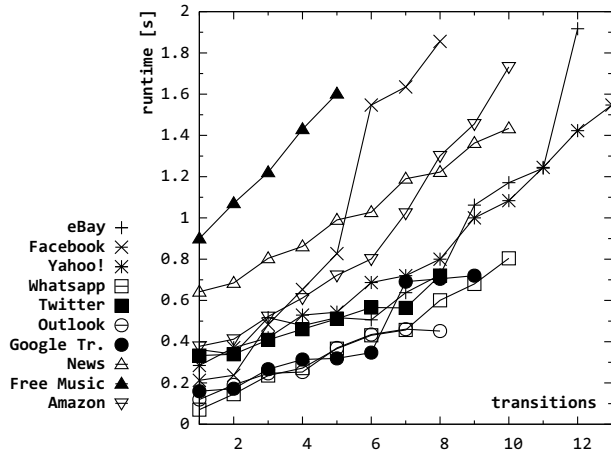


Figure 8: Time taken for generating code vs. number of inter-screen transitions.

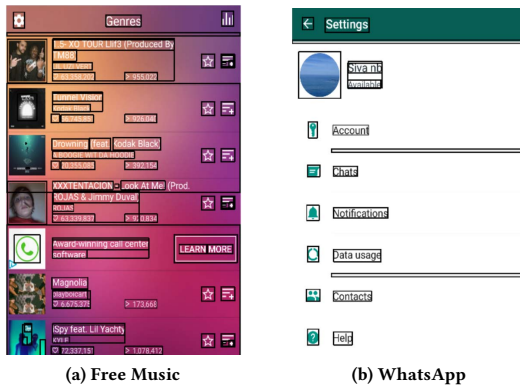


Figure 9: Example inter-screen transition anomalies.

For a given number of transitions, different applications required different amounts of time. A key reason is P2A’s prototype nature. As part of code generation, the prototype parses the P2A-generated layout XML files, whose size depends on the number of inferred screen elements. For example, the Figure 9a Free Music version 1.69 screen has more view elements than the WhatsApp version 2.16.396 screen (Figure 9b). So the P2A-generated Free Music XML file contains more XML nodes. Consequently, it takes P2A longer to again parse the Free Music XML.

Another artifact are the sudden spikes runtime. Similar to differences across apps, this artifact is also caused by widely differing XML parse times due to varying numbers of screen elements.

### 6.3 RQ2: Runtime of In-Screen Animations

Figure 10 shows the time P2A took to infer in-screen animations and generate Android code, broken down by P2A’s major components. P2A’s average runtime was 26 seconds, with a maximum of 69 seconds for Zedge Android version 5.16.5.

From Figure 10 it is clear that step 1 (essentially REMAUI) consumed the most time. This is not surprising, as step 1 requires heavy-weight computer vision and OCR tasks, whereas steps 2 & 3 perform relatively light-weight perceptual hashing and matching. Finally, code generation (step 4) is also relatively light-weight compared to step 1.

As a concrete example, in our experiments we obtained the worst results for Zedge, both in terms of runtime and in terms of screen similarity. Figures 11a and 11b show Zedge’s start and end screens, superimposed with P2A-identified view hierarchies. On these screens P2A took a long time, since the screens (and especially the end state screen) have a large number of view elements. This affects all four steps, since they all depend on the number of view elements.

### 6.4 RQ3: Screen Similarity

To measure how similar P2A-inferred animation screens are to the input screens, we employ the (compared to MSE) relatively new metric structural similarity (SSIM) [50, 51]. SSIM values are in  $[-1; 1]$ , values increase with similarity. An SSIM of 1 means the input images are identical. We use the standard SSIM definition [51], where SSIM is the average of the following local SSIM computed for each position of a sample window sliding over both screens.

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

As usual,  $\mu_a$  is window  $a$ ’s average pixel (intensity) value,  $\sigma_a$  is window  $a$ ’s standard deviation (as a measure of its contrast),  $\sigma_{ab}$  is the correlation of windows  $a$  and  $b$  after removing their averages, and each  $c_i$  is a small constant.

For this experiment, we measured SSIM with the Scikit-Image package. We compared the final state of the original screenshot against the screenshot of the final state reached through the generated code for all 30 applications.

Figure 10 lists all SSIM results. In general the picture similarity was roughly correlated with P2A’s runtime. The more complex the input screen design, the more elements a screen contains, the longer the P2A runtime, and the more opportunities for mistakes.

Returning to our “worst case” example Zedge, Figure 11c shows the screenshot of the final state achieved through the generated animation code. Since the application has a visually dark theme, the identification of view elements was not accurate, resulting in low similarity to the original application’s animation end-state screen.

As another concrete example, for the landing screen animation of Android Chrome version 53.0.2785.135 in Figures 12a and 12b, P2A inferred the desired in-screen animation even though one screen is “grayed out”. For example, when comparing the clock symbols at the bottom of the screen (□ and ■), our prototype computes the perceptual hash value of these images. Their Hamming distance is 4, so below our threshold, and P2A generates a corresponding animation with a relatively high screen similarity.

## 7 RELATED WORK

P2A is implemented on top of REMAUI. A key limitation of REMAUI is that REMAUI can only map a single image to an application



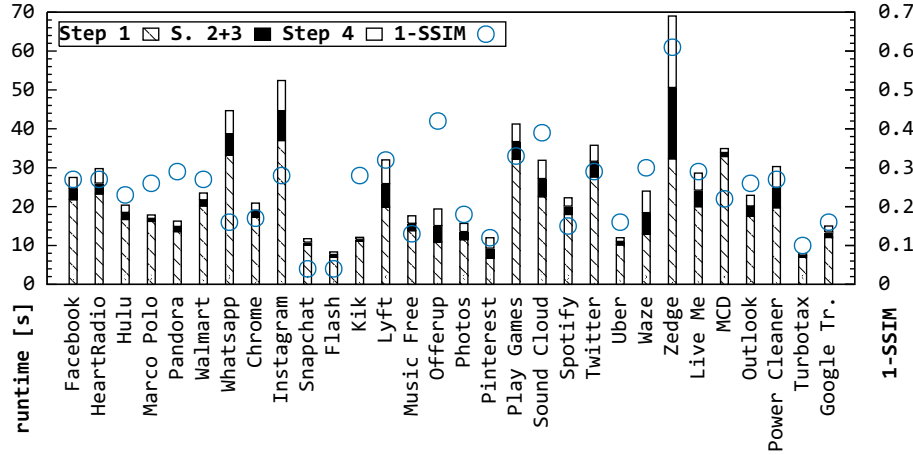


Figure 10: Runtime and similarity for inferring in-screen animations. For both runtime and 1-SSIM, lower values are better. Here step 1 is a REMAUI-style preprocessing pass, steps 2 & 3 match screen elements, and step 4 generates code.



Figure 11: Example on which P2A’s in-screen animation inference performed poorly: Zedge has both a dark visual theme and many view hierarchy elements.

consisting of a single screen. In contrast to REMAUI, P2A takes as input multiple screen designs, allows the user to specify inter-screen transitions, and infers from the screen designs in-screen animations.

While modern IDEs such as Eclipse, Xcode, and Android Studio have powerful interactive builders for graphical user interface (GUI) code [56, 57], IDEs’ prime audience is software developers. Even for its core audience, using such a GUI builder to re-create a complex screen design is a complex task. For example, in an evaluation of GUI builders on a set of small tasks, subjects using Apple’s Xcode GUI builder introduced many bugs that later had to be corrected [57]. Subjects produced these bugs even though the study’s target layouts were much simpler than those commonly found in third-party mobile applications.

Despite much progress in tools for creating user interfaces that combine the unique talents of graphic designers and programmers [9, 34], much conceptual user interface design work is still

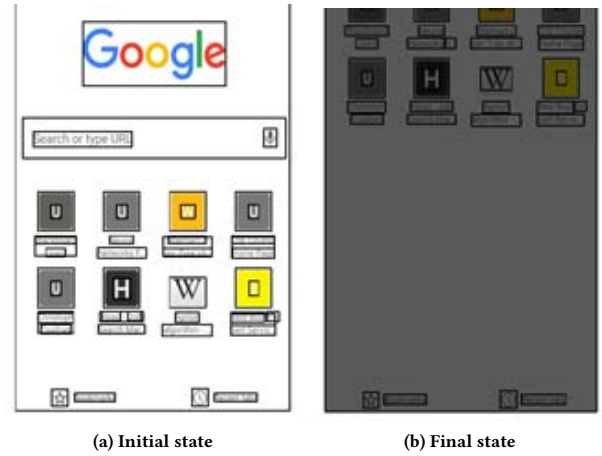


Figure 12: In-screen animation challenge from Figure 6: Android Chrome new tab screen animation.

being done by graphic designers with pencil on paper and digitally, e.g., in Photoshop. Previous work has produced fundamentally different approaches to inferring user interface code, as it was based on different assumptions.

Following are the main changed assumptions for mobile application UI development and reverse engineering that motivate our work. (1) First, many UX designers and graphic artists do not construct their conceptual drawings using a predefined visual language we could parse [5, 10, 28, 42]. (2) Second, while this was largely true for desktop development, mobile application screens are not only composed of the platform’s standard UI framework widgets [13, 14]. (3) Finally, we cannot apply runtime inspection [8, 29] as Pixel-to-App runs early in the development cycle.

Specifically, the closest related work is MobiDev [42], which recognizes instances of a predefined visual language of standard

UI elements. For example, a crossed-out box is recognized as a text box. But unlike Pixel-to-App, MobiDev does not integrate well with a professional mobile application development process. It would require UX designers and graphic artists to change the style of their paper and pencil prototypes, for example, to replace real text with crossed-out boxes. Such changes may reduce the utility of the prototypes for other tasks such as eliciting feedback from project stakeholders. In a traditional reverse engineering setting, MobiDev cannot convert screenshots into UI code.

SILK and similar systems bridge the gap between pen-based GUI sketching and programming of desktop-based GUIs [5, 10, 28]. Designers use a mouse or stylus to sketch directly in the tool, which recognizes certain stroke gestures as UI elements. But these tools do not integrate well with current professional development processes as they do not work on paper-on-pencil scans or screenshots. These tools also do not recognize handwritten text or arbitrary shapes.

UI reverse engineering techniques such as Prefab [13] depend on a predefined model of UI components. The work assumes that the pixels that make up a particular widget are typically identical across applications. However, this is not true for a mobile application UI. Mobile applications often have their own unique, non-standard identity, style, and theme. For Prefab to work, all possible widget styles and themes of millions of current and future mobile applications would need to be modeled.

PAX [8] heavily relies on the system accessibility API at program runtime. At runtime PAX queries the accessibility API to determine the location of text boxes. The accessibility API also gives PAX the text contents of the text box. PAX then applies computer vision techniques to determine the location of words in the text. If a view does not provide accessibility, PAX falls back to a basic template matching approach. PAX thus cannot reverse engineer the UI structure of mobile applications from their screenshots or application design images alone.

Recent work applies the ideas of SILK and DENIM to mobile applications [12], allowing the user to take a picture of a paper-and-pencil prototype. The tool allows the user to place arbitrary rectangles on the scanned image and connect them with interaction events. This idea is also implemented by commercial applications such as Pop for iOS. As SILK and DENIM, this approach is orthogonal to Pixel-to-App.

UI reverse engineering technique used in the tool, Androider [43] solves a problem of porting Graphical User Interface (GUI) from one platform to another. Androider helps in porting UI from Java Swing to Android SDK or even from Android SDK to Objective C. Androider does not reverse engineer the input bitmaps to GUI code but extracts information from the applications in-memory representation using Java Reflection. Unlike Androider, P2A infers in-screen animation from static screen designs.

Among commercial tools, conventional animation creator such as Adobe Flash and Adobe After Effects are more of a design studio. The Flash output is video files displaying each animation present. Reworking these Flash animation videos for every design iteration is very expensive. Since the output deliverable is a video, developers cannot reuse the output deliverable in the development process.

A few start-up companies, i.e., Neonto and PencilCase, have introduced commercial graphic design tools<sup>2</sup> that produce mobile application code. However, these design tools require graphic designers to switch from the designers' current design tools of choice to these new tools. So adopting such a design tool requires graphic designers to abandon the years of experience they have gathered in operating state-of-the-art tools such as Photoshop or to switch from a pencil on paper process to these new digital design tool.

Zeplin [24] increases the reusability of artifacts from the design phase to the development phase. Zeplin allows importing designs from Photoshop or Sketch and extracts the text styles designed by the designers and generates the necessary snippet for layout.xml, styles.xml, and colors.xml style files.

The recent development Supernova Studio [45] goes a step further and converts designs from Sketch to native application code. In contrast, P2A extracts the layout information and styles information from plain bitmap images (pixels) and generates Android code for binding the UI with data. P2A also generates all necessary Java and XML code transitions and animations.

Frameworks such as FramerJS [23] are scripted via Javascript or CoffeeScript. But animation scripts developed for FramerJS are not consumable in the app development process for Android or iOS.

The gap between early prototyping and formal layout definition also exists in the related domain of web site development. A study of 11 designers at 5 companies showed that all designers started with sketching the layout, hierarchy, and flow of web pages with pencil on paper and in graphical design tools such as Photoshop [36].

A similar design process has been reported for desktop applications. At Apple, user interface sketches were first created with a fat marker (to prevent premature focus on details) and later scanned [52]. Separate studies of hundreds of professionals involved in UI design in various companies indicated heavy use of paper-based sketches [6, 27]. One of the reasons was that sketching on paper is familiar due to designers' graphic design background.

## 8 FUTURE WORK

Future work includes replicating the experiments with a broader mix of subjects, beyond the screenshots of the top-100 free Android applications from the Google Play store. Beyond including screenshots of iOS applications, it would also be good to include screen designs, both those produced with professional design tools and scanned pencil on paper sketches.

## 9 CONCLUSIONS

The P2A tool adopts computer vision techniques for developing animated mobile applications. P2A infers from mobile application screen designs the user interface portion of an application's source code and other assets that are ready to be compiled and executed on a mobile phone. Among others, inferred mobile applications contain inter-screen transitions and in-screen animations. In our experiments on screenshots of 30 highly-ranked third-party Android applications, the P2A-generated application user interfaces exhibited high pixel-to-pixel similarity with their input screenshots. P2A took an average of 26 seconds to infer in-screen animations.

<sup>2</sup>Neonto Studio ([www.neonto.com](http://www.neonto.com)), PencilCase Studio ([pencilcase.io](http://pencilcase.io))

## REFERENCES

- [1] Apple Inc. 2013. View programming guide for iOS. <https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewPGPhoneOS/Introduction/Introduction.html>. (Oct. 2013). Accessed March 2018.
- [2] Tanzirul Azim and Iulian Neamtii. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 641–660.
- [3] Patti Bao, Jeffrey S. Pierce, Stephen Whittaker, and Shumin Zhai. 2011. Smart phone use by non-mobile business users. In *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, 445–454.
- [4] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer Vision with the OpenCV Library* (first ed.). O'Reilly.
- [5] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. 2002. JavaSketchIt: Issues in sketching the look of user interfaces. In *Proc. AAAI Spring Symposium on Sketch Understanding*. AAAI, 9–14.
- [6] Pedro F. Campos and Nuno Jardim Nunes. 2007. Practitioner tools and workstyles for user-interface design. *IEEE Software* 24, 1 (Jan. 2007), 73–80.
- [7] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8, 6 (Nov. 1986), 679–698.
- [8] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2011. Associating the visual representation of user interfaces with their internal structures and metadata. In *Proc. 24th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 245–256.
- [9] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe P. Mertz. 2004. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proc. 17th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 267–276.
- [10] Adrien Coyette, Suzanne Kieffer, and Jean Vanderdonck. 2007. Multi-fidelity prototyping of user interfaces. In *Proc. 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, 150–164.
- [11] Tiago Silva da Silva, Angela Martin, Frank Maurer, and Milene Selbach Silveira. 2011. User-centered design and agile methods: A systematic review. In *Proc. Agile Conference (AGILE)*. IEEE, 77–86.
- [12] Marco de Sà, Luís Carriço, Luís Duarte, and Tiago Reis. 2008. A mixed-fidelity prototyping tool for mobile devices. In *Proc. Working Conference on Advanced Visual Interfaces (AVI)*. ACM, 225–232.
- [13] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 1525–1534.
- [14] Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 969–978.
- [15] Leopoldina Fortunati and Sakari Taipale. 2014. The advanced use of mobile phones in five European countries. *The British Journal of Sociology* 65, 2 (June 2014), 317–337.
- [16] Marko Gargenta and Masumi Nakamura. 2014. *Learning Android: Develop Mobile Apps Using Java and Eclipse* (second ed.). O'Reilly.
- [17] Zahid Hussain, Martin Lechner, Harald Milchrahm, Sara Shahzad, Wolfgang Slany, Martin Umgeher, Thomas Vlk, and Peter Wolkerstorfer. 2008. User interface design for a mobile multimedia application: An iterative approach. In *Proc. 1st International Conference on Advances in Computer-Human Interaction (ACHI)*. IEEE, 189–194.
- [18] Flinto Inc. 2017. <https://www.flinto.com/>. (2017). Accessed March 2018.
- [19] Google Inc. 2017. Material design for Android. <https://developer.android.com/design/material/index.html>. (2017). Accessed March 2018.
- [20] Google Inc. 2017. Top free in Android apps. <https://play.google.com/store/apps/collection/topselling?hl=en>. (2017). Accessed March 2018.
- [21] Invision Inc. 2017. <https://www.invisionapp.com/>. (2017). Accessed March 2018.
- [22] Kony Inc. 2014. Bridging the gap: Mobile app design and development. <http://forms.kony.com/rs/konyolutions/images/BridgingGapBrochuredec1014.pdf>. (Dec. 2014). Accessed March 2018.
- [23] Motif Tools BV Inc. 2017. <https://framer.com/>. (2017). Accessed March 2018.
- [24] Zeplin Inc. 2017. <https://www.zeplin.io/>. (2017). Accessed March 2018.
- [25] Amy K. Karlson, Brian Meyers, Andy Jacobs, Paul Johns, and Shaun K. Kane. 2009. Working overtime: Patterns of smartphone and PC usage in the day of an information worker. In *Proc. 7th International Conference on Pervasive Computing (Pervasive)*. Springer, 398–405.
- [26] Kati Kuusinen and Tommi Mikkonen. 2013. Designing user experience for mobile apps: Long-term product owner perspective. In *Proc. 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 535–540.
- [27] James A. Landay and Brad A. Myers. 1995. Interactive sketching for the early stages of user interface design. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 43–50.
- [28] James A. Landay and Brad A. Myers. 2001. Sketching interfaces: Toward more human interface design. *IEEE Computer* 34, 3 (March 2001), 56–64.
- [29] Xiaojun Meng, Shengdong Zhao, Yongfeng Huang, Zhongyuan Zhang, James Eagan, and Ramanathan Subramanian. 2014. WADE: simplified GUI add-on development for third-party software. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 2221–2230.
- [30] Mehmet Kivanç Mihçak and Ramarathnam Venkatesan. 2001. New iterative geometric methods for robust perceptual image hashing. In *Proc. 1st ACM Digital Rights Management Workshop (DRM)*. Springer, 13–21.
- [31] Vishal Monga and Brian L. Evans. 2006. Perceptual image hashing via feature points: Performance evaluation and tradeoffs. *IEEE Transactions on Image Processing* 15, 11 (Nov. 2006), 3452–3465.
- [32] Brad A. Myers. 1994. Challenges of HCI design and implementation. *Interactions* 1, 1 (Jan. 1994), 73–83.
- [33] Brad A. Myers. 2012. Graphical user interface programming. In *Computer Science Handbook* (second ed.), Allen B. Tucker (Ed.). CRC Press.
- [34] Brad A. Myers, Scott E. Hudson, and Randy F. Pausch. 2000. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* 7, 1 (March 2000), 3–28.
- [35] Vanda Nahavandipoor. 2013. *iOS 7 Programming Cookbook* (first ed.). O'Reilly.
- [36] Mark W. Newman and James A. Landay. 1999. *Sitemaps, storyboards, and specifications: A sketch of web site design practice as manifested through artifacts*. Technical Report UCB/CSD-99-1062. EECS Department, University of California, Berkeley.
- [37] Tuan A. Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–259.
- [38] Greg Nudelman. 2013. *Android Design Patterns: Interaction Design Solutions for Developers*. Wiley.
- [39] R. Plamondon and S.N. Srihari. 2000. Online and off-line handwriting recognition: A comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22, 1 (Jan. 2000), 63–84.
- [40] James R. Rudd, Kenneth R. Stern, and Scott Isensee. 1996. Low vs. high-fidelity prototyping debate. *Interactions* 3, 1 (1996), 76–85.
- [41] Seyyed Ehsan Salamat Tabataba, Iman Keivanloo, Ying Zou, Joanna Ng, and Tinny Ng. 2014. An Exploratory Study on the Relation between User Interface Complexity and the Perceived Quality of Android Applications. In *Proc. 14th International Conference on Web Engineering (ICWE)*. Springer.
- [42] Julian Seifert, Bastian Pfleging, Elba del Carmen Valderrama Bahamóndez, Martin Hermes, Enrico Rukzio, and Albrecht Schmidt. 2011. Mobidev: A tool for creating apps on mobile phones. In *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, 109–112.
- [43] Eeshan Shah and Eli Tilevich. 2011. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In *Proc. Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH Workshops)*. ACM, 255–260.
- [44] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps. In *Proc. ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS)*. ACM, 275–284.
- [45] Supernova Studio. 2018. <https://supernova.studio/>. (2018). Accessed March 2018.
- [46] Ashwin Swaminathan, Yinian Mao, and Min Wu. 2006. Robust and secure image hashing. *IEEE Transactions on Information Forensics and Security* 1, 2 (June 2006), 215–230.
- [47] Richard Szeliski. 2010. *Computer Vision: Algorithms and Applications*. Springer.
- [48] The Nielsen Company. 2013. The Mobile Consumer: A Global Snapshot. <http://www.nielsen.com/us/en/insights/reports/2013/mobile-consumer-report-february-2013.html>. (Feb. 2013). Accessed March 2018.
- [49] Øivind Due Trier, Anil K. Jain, and Torfinn Taxt. 1996. Feature extraction methods for character recognition—A survey. *Pattern Recognition* 29, 4 (April 1996), 641–662.
- [50] Zhou Wang and Alan C. Bovik. 2009. Mean squared error: Love it or leave it? A new look at signal fidelity measures. *IEEE Signal Processing Magazine* 26 (Jan. 2009), 98–117. Issue 1.
- [51] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (April 2004), 600–612.
- [52] Yin Yin Wong. 1992. Rough and ready prototypes: Lessons from graphic design. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), Posters and Short Talks*. ACM, 83–84.
- [53] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proc. 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 89–99.
- [54] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static window transition graphs for Android. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 658–668.
- [55] Christoph Zauner. 2010. *Implementation and benchmarking of perceptual image hash functions*. Master's thesis. Upper Austria University of Applied Sciences,

- Hagenberg Campus.
- [56] Clemens Zeidler, Christof Lutteroth, Wolfgang Stürzlinger, and Gerald Weber. 2013. The Auckland layout editor: An improved GUI layout specification process. In *Proc. 26th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 343–352.
- [57] Clemens Zeidler, Christof Lutteroth, Wolfgang Stürzlinger, and Gerald Weber. 2013. Evaluating direct manipulation operations for constraint-based layout. In *Proc. 14th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, 513–529.