Toward a deeper integration of low-fidelity sketches into mobile application development

by

SOUMIK MOHIAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2023

Copyright © by Soumik Mohian 2023

All Rights Reserved

To my father and mother, whose love and support have been the driving force behind everything I have achieved.

ACKNOWLEDGEMENTS

To begin with, I would like to express my heartfelt appreciation to my supervising professor Dr. Christoph Csallner. Without his unwavering support, this research project would not have been possible. I am very fortunate to have had such a remarkable mentor, and I cannot thank him enough for his invaluable assistance throughout my doctoral studies. I greatly thank my Ph.D. committee members, Dr. Jean Gao, Dr. William Beksi, and Dr. Won Hwa Kim, for their constant support and constructive feedback during my entire doctoral journey. Lastly, I would like to acknowledge the help of my family. I thank my wife, Farzana Akter, for her unwavering support throughout this journey. I want to express my appreciation to my mother and father, whose love, support, and guidance have played an instrumental role in shaping the person I am today. Their constant encouragement and belief in me have been a source of strength and inspiration throughout my life.

April 19, 2023

ABSTRACT

Toward a deeper integration of low-fidelity sketches into mobile application development

Soumik Mohian, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professor: Dr. Christoph Csallner

Mobile application development often starts with creating low-fidelity sketches of user interfaces. Integrating these sketches into the software development process can reduce repetition, narrow the gap between user perception and final implementation, and improve app resilience. In this study, we introduce the DoodleUINet dataset, which comprises over 10K sketches of UI elements. Our Doodle2App tool converts low-fidelity sketches into a single-page, compilable Android app. At the same time, our PSDoodle provides an interactive, partial sketch-based search engine with a top-10 screen retrieval accuracy comparable to the state-of-the-art SWIRE line of work but with a 50% reduction in the average required time. Our TpD tool leverages natural language and sketching to retrieve and display the target screen in its top-10 search results with a success rate of 90%, achieving the top-10 accuracy of the state-of-the-art approach and consistently showing the target screen in the top-30. We have also developed D2S2, a drag-and-drop-based mobile screen search tool that incorporates the backend of TpD. Overall, our tools provide valuable resources for novice software engineers and facilitate the integration of low-fidelity sketches into the software development process.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iv

ABSTRACT . . . v

Chapter Page

1. Introduction . . . 1

2. Doodle2App: Native App Code by Freehand UI Sketching . . . 7

2.1 Introduction . . . 7

2.2 Background . . . 10

2.2.1 Sketch to Code with Teleport . . . 10

2.2.2 Recurrent Neural Networks (RNNs) . . . 11

2.3 UI Elements & Sketch Samples . . . 11

2.4 Overview and Design . . . 13

2.4.1 RNN-based UI Element Sketch Classifier . . . 14

2.4.2 Generating UI Code . . . 15

2.5 Preliminary Micro Evaluation . . . 16

2.5.1 RQ1: Fast Classification & Conversion . . . 16

2.5.2 RQ2: More Accurate Than Teleport . . . 17

2.6 Conclusions . . . 17

3. PSDoodle: Fast App Screen Search via Partial Screen Doodle . . . 19

3.1 Introduction . . . 19

3.2 Background . . . 22

3.2.1 SWIRE: Offline Full-screen Search . . . 23

3.2.2 Google QuickDraw & DoodleUINet . . . 23

3.3 Overview and Design 25
3.3.1 Rico Screens & UI Element Labels 25
3.3.2 Query Language: Stylized + Flexible Doodles 27
3.3.3 UI Element Doodle Recognition 29
3.3.4 Searching Screens for UI Element Doodles 31
3.3.5 Hyperparameter Optimization 34
3.4 Evaluation 36
3.4.1 RQ1: Recognizing Partial Icon Doodles 38
3.4.2 RQ2: Top-10 Screen Search Accuracy 39
3.4.3 RQ3: Search With Partial-screen Sketches 41
3.4.4 RQ4: Interactive and Fast Screen Retrieval 42
3.4.5 High-level Feedback from Participants 43
3.4.6 Relaxing PSDoodle's Query Language 44
3.5 Related Work 44
3.6 Conclusions 46
4. PSDoodle: Searching for App Screens via Interactive Sketching 47
4.1 Introduction 47
4.2 Background 48
4.3 Overview and Design 51
4.3.1 PSDoodle's Website 51
4.3.2 Recognizing Individual UI Elements 53
4.4 Exploring PSDoodle's Usage 55
4.4.1 Supporting Different Sketching Styles 55
4.4.2 Surfacing Several Relevant Result Screens 57
4.4.3 User Experience 57
4.5 Conclusions 58

5. Searching Mobile App Screens via Text + Doodle 59
5.1 Introduction 59
5.2 Background 62
5.2.1 Rico: Corpus of 72k Android App Screens 62
5.2.2 Swire: Search by Full-screen Sketch 64
5.2.3 PSDoodle: Search by Icon Doodles 64
5.3 Overview and Design 65
5.3.1 Screen Search Via Icon Doodles & Positional Text Queries 66
5.3.2 Screen Texts, UI Descriptions, And X/Y Pixel Coordinates 68
5.3.3 Screen Contents' Stop Words, Names, And Lemmatization 68
5.3.4 Adding Screen Content Synonyms 70
5.3.5 Storing Screen Contents' Locations And Synonyms 72
5.3.6 Drawing Recognition With TpD 73
5.3.7 Ranking Mobile Screens 75
5.4 Evaluation 77
5.4.1 RQ1: Achieving state-of-the-art performance in Top-10 Accuracy . 79
5.4.2 RQ2: Efficient and Interactive Screen Retrieval 79
5.4.3 RQ3: Displaying Multiple Relevant Screens 81
5.4.4 RQ4: Improving Search Relevance via Text & Sketch 81
5.4.5 Screen Searches Ranked Outside Top-10 82
5.4.6 User Feedback 83
5.5 Conclusions 84
6. D2S2: Drag 'n' Drop Mobile App Screen Search 86
6.1 Introduction 86
6.2 Background 88
6.3 Overview and Design 89

6.3.1 User Interface & Query Language 89
6.3.2 D2S2's Back-end 92
6.4 D2S2 usage 94
6.4.1 Similar Screen Search Performance as TpD 94
6.4.2 More Targeted Than Google Image Search 96
6.5 Conclusions 96
7. Conclusions 98
Appendix
REFERENCES 99

CHAPTER 1

Introduction

Sketching plays a significant role in mobile application development [1, 2, 3]. Sketching provides a visual medium for conveying design ideas, visualizing abstract concepts, and comparing alternatives. Designers often use sketches to create rough images of screen layouts during the initial phase of designing user interfaces. Unlike other tools used in the development process, sketching has a low barrier to entry as it doesn't require specialized software or technical skills. One of the most significant benefits of sketching is that it can help designers identify potential usability issues early in the design process. Sketching allows them to make adjustments before investing time and resources in the development of the application, saving valuable time and resources while enhancing the overall user experience.

Mobile app development is rapidly growing due to its widespread and increasing use in all organizations, including social, industrial, educational, and small businesses [4, 5]. Developing a fully functional app requires a team of experts with a spectrum of skills to meet the demand. There is a significant disconnect between users' perceptions, designers' attempts to translate that into the user interface, and software developers' final implementation using formal programming, resulting in inconsistency between the intended design and the final product. This study bridges the knowledge gap by leveraging low-fidelity ubiquitous sketches and creating various tools. Specifically, the study examines the potential of drawings to mitigate the burden of repetitive and low-level technical details associated with app development. Furthermore, the study investigates the efficacy of sketches that can

empower novice software developers to develop more resilient real-world mobile applications.

Deep learning has exhibited notable achievements in numerous domains, particularly in image processing, where it has demonstrated high success rates[6, 7]. The remarkable achievements of deep learning are attributed mainly to the development of the ImageNet dataset [8] comprises thousands of labeled images of ordinary objects. Deep learning methods can leverage stroke sequence data for sketch recognition [9]. QuickDraw has approximately 50 million doodles belonging to 345 common categories, and only a few of these categories are applicable for UI sketching [10, 11].We created the DoodleUINet dataset, collecting over 10k sketches associated with 16 common UI element categories via Amazon Mechanical Turk [12]. The DoodleUINet dataset fills this gap, providing a valuable resource for researchers and developers.

Doodle2App [13] was our initial endeavor to incorporate low-fidelity sketches of QuickDraw and DoodleUINet into the development of mobile applications. Specifically, Doodle2App employs a bi-directional RNN network to train on both QuickDraw and DoodleUINet datasets, allowing it to accurately identify UI element categories based on the strokes on the digital interface. After correctly classifying a new UI element, Doodle2App deals with overlapping and nesting. Doodle2App's website offers a preview of the UI design produced by the sketch. Furthermore, Doodle2App generates a compilable single-page Android application with layout codes and resource files with images and styles. The resulting application includes basic interactions with the preview and generated code. The performance of Doodle2App exceeds that of the state-of-the-art tool Teleport, as it achieved a higher level of accuracy in classifying UI element categories.

For further integration of sketch in the software development process, PSDoodle offers a sketch-based mobile screen search engine for novice software developers [14, 15]. This tool overcomes the limitations of keyword searches and other shortcomings associated

with sketch-based solutions by providing an interactive, iterative tool of its first kind. Using a Long short-term memory(LSTM) network trained on both the QuickDraw and DoodleUINet datasets, PSDoodle can accurately identify UI categories from sketches drawn on a digital interface. PSDoodle is built on top of a subset of the Rico repository [16]. By leveraging a sketched query's UI element type, position, and element shapes, PSDoodle fetches real-world UI examples from the Rico dataset. PSDoodle achieves a top-10 screen retrieval accuracy similar to the state-of-the-art while cutting the average search time in half.

TpD [17] designed to enhance the efficiency of mobile screen search by integrating a keyword-based search with PSDoodle. Notably, TpD is the first to offer interactive and iterative mobile screen searches by combining text and sketch input. Its approach entails consideration of the displayed texts on the mobile screen and UI element descriptions, alongside their position, to identify relevant mobile screens based on the text query. The tool blends several Natural Language techniques to enhance the retrieval performance. Novice software engineers conducted evaluations revealing that TpD outperforms PSDoodle in top-10 accuracy and achieves comparable results to Swire. The target screen consistently appears within the top-30. Additionally, TpD demonstrates improved efficiency by requiring less time than its predecessor, PSDoodle.

Our latest research endeavor, D2S2, capitalizes on the TpD backend and introduces a novel approach to mobile screen search utilizing drag-and-drop functionality as an alternative to the sketch-based methods employed in PSDoodle. D2S2 converts the dragged UI element into a sketched representation of PSDoodle, extracts all texts added on the search interface, and passes the query to the TpD' algorithm to find the top 20 mobile app screens from Rico. D2S2 updates the search result in the Top-pick section each time a user adds, removes, resizes, and moves UI elements on the canvas of D2S2. We recruited ten software

developers to evaluate the performance of D2S2. D2S2 achieves comparable performance and shows more relevant screens than Google Image Search and TpD.

In summary, this dissertation presents novel contributions to the integration of low-fidelity sketches into mobile application development, which are as follows:

- Doodle2App generates a preview of the user interface and a compilable single-page Android application from low-fidelity digital drawings (Chapter 2).
- We developed PSDoodle, designed to facilitate the search for real-world mobile applications using low-fidelity sketches. To our knowledge, PSDoodle is the first tool to offer interactive and iterative search capabilities using partial drawings. (Chapters 3 and 4).
- TpD combines keyword-based search functionality with sketching and enhances the speed and accuracy of searching for real-world mobile applications (Chapter 5).
- Leveraging the back-end capabilities of TpD, we have developed D2S2, a drag-and-drop-based interactive mobile screen search tool (Chapter 6).

The research studies offer open-source licenses to access the source code, processing scripts, training data, and experimental results. These tools are hosted online and provided to the public free of charge for their usage.

Author Contributions

- **Chapter 2**: Doodle2App: Native App Code by Freehand UI Sketching.

  Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 81-84 2020

  Authors Soumik Mohian, Christoph Csallner

  Dr. Christoph Csallner supervised the entire project, from data collection to tool development. He contributed to developing the research questions and evaluation

strategy, significantly strengthening the paper. I designed the interface to gather and arrange data, conducted the training of deep neural networks, and developed the tool.

- **Chapter 3**: PSDoodle: Fast App Screen Search via Partial Screen Doodle.
  Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 89-99. 2022
  Authors Soumik Mohian, Christoph Csallner
  Dr. Christoph Csallner supervised the research direction, reviewed the research questions and experimentation setup, and significantly improved the paper. My responsibilities entailed the development of both the tool and algorithm, the recruitment of participants for evaluation, and the subsequent facilitation of the evaluation sessions.

- **Chapter 4**: PSDoodle: Searching for App Screens via Interactive Sketching

  Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 84-88. 2022
  Authors Soumik Mohian, Christoph Csallner I designed the web interface, developed the tool, and deployed it on Amazon AWS. Dr. Christoph Csallner offered continuous feedback, contributed to developing research questions, and significantly enhanced the quality of the paper.

- **Chapter 5**: Searching Mobile App Screens via Text + Doodle (The manuscript has been submitted for peer review and is currently available as a preprint)
  Authors Soumik Mohian, Christoph Csallner
  Dr. Christoph Csallner provided invaluable feedback to the development of the user interface, contributed to the setup of experiments, and substantially enhanced the

quality of the paper. My contributions encompassed algorithm implementation, tool interface design, tool deployment on the Amazon Web Services platform, participant recruitment for evaluation purposes, and facilitation of the evaluation sessions.

- **Chapter 6**: D2S2: Drag 'n' Drop Mobile App Screen Search (paper submitted for review)

  Authors Soumik Mohian, Tony Tang, Tuan Trinh, Don Dang, Christoph Csallner

  Tony Tang, Tuan Trinh, and Don Dang were responsible for implementing the web interface, recruiting participants for evaluation, and conducting user studies. Dr. Christoph Csallner supervised the research direction, actively contributing to developing research questions and substantially improving the paper. My specific contribution encompassed developing the back-end to rank screens based on UI elements on the canvas, writing the paper, and drafting the user study.

CHAPTER 2

Doodle2App: Native App Code by Freehand UI Sketching

2.1 Introduction

User interface development of many apps starts with freehand sketching, typically with pen on paper [1, 2, 18, 19, 20]. Integrating such *freehand sketching* more tightly into the software development process is a long-standing research challenge. The long-term goal is to directly convert designers' freehand on-paper sketching to ready-to-compile app code. As a step toward this goal, in this paper we replace pen and paper with mouse or touchpad.

While much progress has been made toward our sub-goal of supporting computer-based sketching (e.g., with a mouse), existing approaches such as SILK and Teleport are still limited [21, 22]. Some only support a few sketch primitives (e.g., ellipse, rectangle, straight line, and squiggly line) and thus support few user-app interaction styles [1, 21, 23, 24, 25, 26]. The others have low recognition accuracy [22].

Tightly integrating freehand user interface (UI) sketching would bridge a significant gap in today's software development process. Specifically, after iterating on paper-based prototypes, UI designers today have to manually recreate prototypes in "high fidelity" tools such as Photoshop or in an IDE, which is laborious and costly (even with GUI builders). This gap also decouples UI designers from the code their team eventually produces, which often leads to UI designs that are hard to implement in programming constructs.

Integrating freehand UI sketching is hard if we aim to maintain paper's well-known benefits, e.g., as documented in a study of 87 CHI attendees who had experience creating or testing UI prototypes [20]. To create UI prototypes the top used tools (72/87) were art

supplies (i.e., paper), because they make it both quick and easy to create and use prototypes and because they promote creativity. For usability studies, the top tool was also art supplies, because they facilitate discussion, allow quick changes, and yield good feedback.

The common limitation of existing computer-based freehand sketching approaches is their reliance on traditional image classification techniques, which do not scale to many sketch primitives or have low accuracy. Computer vision has seen tremendous progress over the last years, especially in deep neural networks that have been trained on large sample collections.

The first key insight is that in image classification generally, having more training samples that are correctly labeled (e.g., "this is a rectangle") enables both distinguishing between more image classes (e.g., rectangle vs. ellipse) and doing so more accurately.

This is true even if the target image classes have little overlap in features with the training samples. The second insight is that humans create a sketch as a sequence of strokes (e.g., with their pen or computer mouse) and to sketch a given object many designers produce a similar stroke sequence (e.g., a rectangle as a single counter-clockwise stroke starting from left-top). So recognizing a stroke sequence is easier than recognizing a final sketch.

At the core of our approach is Google's Quick, Draw! ("QuickDraw") collection of over 50M labeled sketches of 345 categories, from "aircraft carrier" to "zigzag", each given as a stroke sequence [10]. While this sample collection is crucial for high accuracy, it only contains 4 classes we considered relevant for UI design. We, therefore, collected some 12k sketches of 16 UI specific classes via Amazon Mechanical Turk. While the resulting Doodle2App tool is just an early prototype, it already supports several times more classes of graphical primitives than the earlier work on SILK, while being more accurate than the current state of the art tool Teleport (94% vs. 17%). At the same time, by adding to the 12k UI element sketch collection, the Doodle2App approach promises to scale well to more

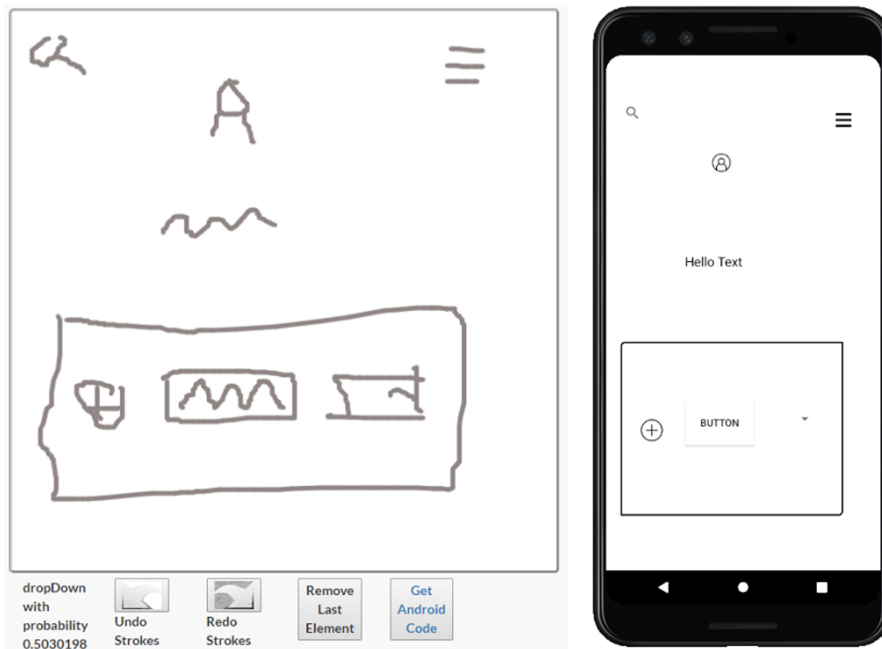


Figure 2.1. Drawing interface and generated Android app..

than 20 UI element classes, as its architecture is very similar to QuickDraw, which already supports over 300 categories at high accuracy.

The Doodle2App tool is currently web-based and provides a canvas for sketching via touchscreen or mouse (Figure 2.1). The user creates one sketch (or "doodle") of a UI element at a time (e.g., a container, text, menu button, forward button, etc.), Doodle2App classifies the element via its recurrent neural network, and correspondingly updates an HTML-based preview of the resulting UI. This gives the designer immediate feedback on the UI construction process. At any time the designer can tell Doodle2App to export the current UI as source code that is ready to compile and run as a single-page app on stock Android devices. To summarize, this paper makes the following major contributions.

- This paper provides the first accurate conversion of freehand UI sketches that have a substantial variety of UI elements.

- To evaluate the approach, the paper implements the novel Doodle2App tool and compares it to the state-of-the-art sketch to code conversion tool Teleport.
- The tool is freely available at `http://pixeltoapp.com/doodle/`.

2.2 Background

This section contains necessary background information on sketch recognition, the state-of-the-art UI sketch to code conversion tool Teleport, and recurrent neural networks (RNNs).

An offline approach processes a finished sketch (e.g., as REMAUI [27] or Teleport [22]). In contrast, an online approach processes a sketch's strokes in the order they are drawn (e.g., as SILK [21]). Offline recognition provides additional use cases (e.g., historical sketches), whereas online recognition has access to more information and thus promises higher accuracy.

Existing approaches also differ in the most basic (aka "atomic" or "primitive") graphical elements they recognize. For example, SILK recognizes four atomic elements (ellipse, rectangle, straight line, and squiggly line). Some approaches then recognize certain atomic element combinations as a compound element (e.g., SILK considers a small box in a long rectangle a slider). While a compound element is still a single UI element, some approaches also support nesting, e.g., a primitive rectangle may contain two other primitives, which are then considered three UI elements, a container with two children.

2.2.1 Sketch to Code with Teleport

The most closely related approach is Teleport's vision API v2 [22]. It supports 21 classes of hand-drawn UI element sketches. Doodle2App's atomic UI element classes (Figure 2.2) overlap with Teleport's in the sense that the respective example sketches on the Teleport website look like our samples. This overlap is squiggle (text), square (which

Doodle2App treats as a container, Teleport as a text area or container), checkbox, switch (toggle), star (rating), dropdown, and slider.

Teleport works offline. To adjust for users' varying light conditions, background noise, camera alignment, and paper skew and rotation, Teleport employs a sophisticated computer vision pipeline. It then classifies elements with a convolutional neural network (CNN). The Teleport website reports an experiment that yielded 85% accuracy, but describes this number as "optimistic".

2.2.2 Recurrent Neural Networks (RNNs)

State-of-the-art approaches for image recognition typically use deep learning and especially convolutional neural networks (CNNs) [6]. CNN assumes that training data is of fixed dimension and independent from each other. For online sketch detection this is a problem, as sketch strokes are ordered and vary in their edge counts. Recurrent neural networks (RNNs) support both of these sketch properties [28]. Doodle2App builds on QuickDraw's network architecture to leverage its recent sketch recognition success [10]. QuickDraw uses bi-directional RNNs [29], which use both stroke sequences and reverse stroke sequences.

2.3 UI Elements & Sketch Samples

Doodle2App currently focuses on Android. The Rico dataset [16] assembled 66k unique Android app screens from 9.3k apps from 27 app categories of the Google Play app store. Rico also captured the runtime UI hierarchy of each screen and clustered all screens' elements by visual similarity. The Rico clusters thus do not only represent the base Android elements but also UI elements of third-party apps. We calculate the occurrence of each Rico-inferred element cluster by parsing all screen hierarchies. According to Rico, the most common Android UI element type was **container**, followed by (in order) **image**, icon

(a small interactive image), **text**, **text button**, web view, input, list item, **switch** (a toggle element), map view, **slider**, and **checkbox**. Rico further breaks down the most common icon (#3 in the above list) types as **back**, followed by **menu** (the hamburger), **cancel** (close), **search** (loupe), **plus** (add), avatar (user image), home (house), **share**, **settings** (gear), **star**, edit, more, refresh, and **forward**.

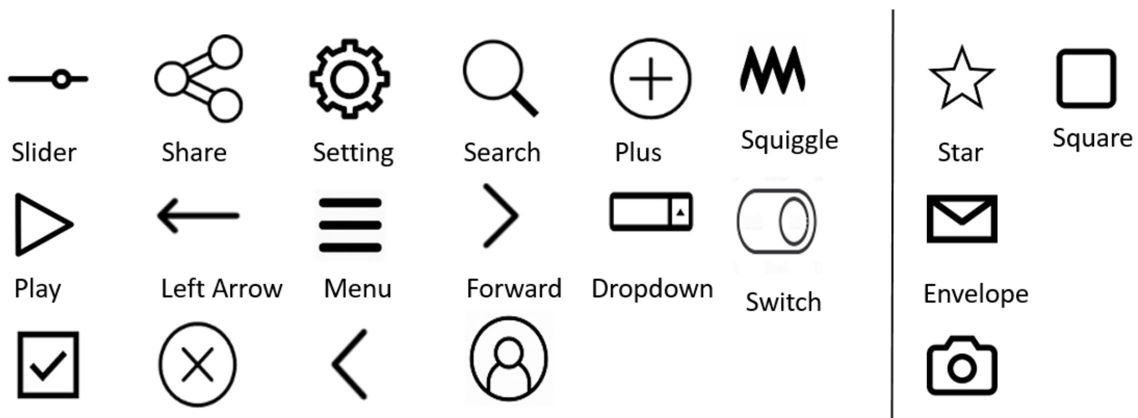


Figure 2.2. The 20 graphical primitives Doodle2App currently recognizes. Samples for 4 classes (right) are from QuickDraw, the rest (left) is from Mechanical Turk..

To demonstrate the flexibility of the approach, Doodle2App supports several of these top UI elements (i.e., the boldfaced ones above), mostly as graphical primitives (Figure 2.2). Besides primitives, Doodle2App also supports an example compound element: A squiggle (text) that fills most of a rectangle is a text button. Finally, Doodle2App supports nested elements, i.e., a rectangle is a container that can contain several other elements.

Combining Doodle2App with optical character recognition for text detection is future work, e.g., by adding an OCR engine. As a work-around, Doodle2App currently treats a squiggly line as text. Further work-arounds deal with detecting arbitrary images and treating an avatar image as an arbitrary image.

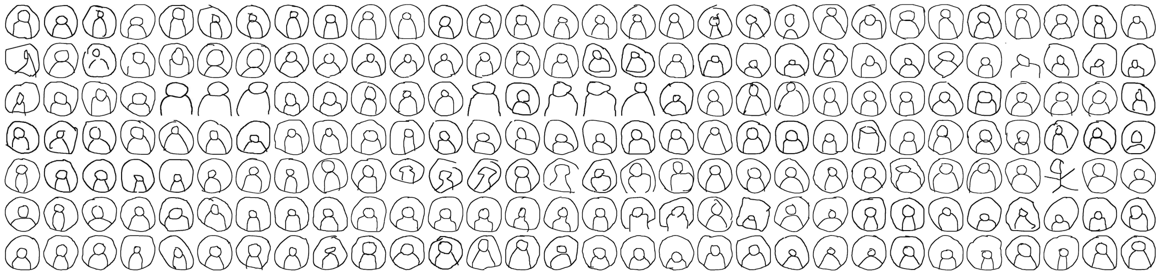


Figure 2.3. Subset of DoodleUINet's freehand avatar sketches. Each sketch is a stroke sequence, each stroke a line sequence. .

The QuickDraw dataset [10] contains 345 sketch categories (from "aircraft carrier" to "zigzag"), with some 100k samples each, drawn by anonymous users [11]. QuickDraw stores each sketch as a sequence of strokes. Each stroke is a sequence of straight lines, given by their x/y endpoint coordinates.

To collect sketches for the 16 remaining categories from Figure 2.2 we built a website similar to Sketchy [30]. To encourage users to draw from memory, our website shows a UI element repeatedly for one second before blacking it out for 5 seconds. We recruited participants via Amazon Mechanical Turk (with IRB approval) and asked each participant to produce 15 sketches of graphical primitives. We thereby collected 11,500 drawings. After manual review, each of our 16 categories contained some 600 sketches (such as the avatar sketches in Figure 2.3).

2.4 Overview and Design

Doodle2App currently side-steps the complications of capturing designers' paper-based freehand sketching activities. Instead, a designer directly sketches on Doodle2App's website via mouse or touchscreen (Figure 2.1 left). Since a UI element can consist of several strokes, the designer draws one UI element at a time and indicates the end of one

UI element by pressing the “z” key or double-tapping the canvas. To give the designer immediate feedback, the website also shows an interactive HTML-based preview.

Doodle2App currently supports 21 UI element types, the 20 primitives from Figure 2.2 plus the compound text button. On a double-tap, Doodle2App passes the collected strokes to its custom RNN-based UI element classifier, resolves overlap and nesting, and updates its HTML preview. At any time the designer can export the current UI state to a compilable Android app.

2.4.1 RNN-based UI Element Sketch Classifier

Since deep learning works best when training samples are equally distributed over the classes, we only used a small subset of the samples from our four QuickDraw classes, i.e., some 600 (instead of the full 100k). We selected these 2.4k samples by manually reviewing the first samples from QuickDraw, rejecting clear outliers.

Since deep learning has millions of parameters and works best with larger training sets [31], we used transfer learning [32] to benefit from a network that has been pre-trained on more samples. Specifically, we randomly picked 20 QuickDraw classes outside our 4 QuickDraw classes, split their 2M samples into training and test (80/20), normalized and converted them into TensorFlow’s binary storage format tfrecords [33], trained the existing QuickDraw network architecture [10] on the training set, and initially achieved 94% accuracy on the test set.

Since our application has 20 classes (vs. 345 in QuickDraw) we changed the network architecture, to pick up additional subtleties in the training set and thus improve accuracy. The resulting architecture consists of a convolutional neural network (CNN) layer (with filter size 5, kernel size 48), followed by a CNN layer (5, 64), another CNN layer (3, 96), 8 Bi-RNN layers (as opposed to 5 in QuickDraw), and a fully-connected layer. We trained

this network for 155,138 steps with a batch size of 8. Overall, adding 3 bidirectional RNN (Bi-RNN) layers to QuickDraw's 5 existing Bi-RNN layers increased accuracy to 98%.

We further trained this pre-trained network with 80% of our 12k sample set for 32,500 steps with a batch size of 8. Our complete sample set is available both at the stroke sequence level and as visualizations. This yielded 96.1% accuracy on our test dataset of 2.4k samples [12].

2.4.2 Generating UI Code

After classifying a new UI element, Doodle2App deals with overlap and nesting. If the element overlaps with a rectangle (container), then Doodle2App moves it either inside or outside the container, depending on the overlap. If the new element is a squiggle (text) and takes more than half of the container, Doodle2App considers it a text button, otherwise it becomes a nested element. If the new element overlaps with a non-container element, Doodle2App disregards the element if the overlap is greater 50%, otherwise it moves the new element outside the area of the existing element.

The nesting relation defines the app screen's UI hierarchy. Doodle2App creates a compilable single-page Android app, complete with the UI hierarchy's layout code and resource files for style and images. Doodle2App rescales element positions into default Android screen resolution. Figure 2.1 (right) shows an example generated Android app. In both preview and app the buttons and checkbox are clickable, dropdowns have sample items, and sliders and toggles show state change on click. Both preview and generated app use a basic interactive graphical representation of the detected elements. Inferring custom element styles is future work.

2.5 Preliminary Micro Evaluation

To gauge the potential of Doodle2App, we performed an initial evaluation at the micro-benchmark level, i.e., at the level of recognizing and converting individual atomic UI elements. We consider this a necessary first step, as without good micro-level performance it is unlikely the technique will do well in the more complex whole-screen or whole-app setting. We thus evaluate Doodle2App in terms of runtime and precision, and compare it with the most closely related competitor, Teleport, using the following research questions.

RQ1 What is Doodle2App's runtime to classify a UI element sketch and convert it to Android code?

RQ2 How does Doodle2App compare with the state-of-the-art tool Teleport in terms of classification accuracy?

We trained our classifier with an 8 GB RAM Nvidia GeForce GTX 1080 GPU on a local 16 GB RAM 64-bit Windows 10 machine with a 3.4 GHz Intel i7-6700 CPU. We first trained our network for some 62 hours on the 20 random QuickDraw categories for 155,138 steps. Then we continued training the network on our dataset for 32,500 steps, for another 18 hours, for a total of some 80 hours.

2.5.1 RQ1: Fast Classification & Conversion

For a preliminary exploration of Doodle2App's runtime we sketched and processed some 20 atomic UI elements, both locally on a 16 GB RAM 64-bit Windows 10 machine with a 2.20 GHz Intel i7-8750H CPU and on an AMD64 Ubuntu 16.04.5 Amazon EC2 t2.micro instance. The average runtime to process and classify a UI element sketch was 26 ms (locally) and 20 ms (EC2). These times include neither transmission delays between user and EC2 nor the time it took to update Doodle2App's interactive HTML preview.

After detecting a UI element drawn on the interface, the average runtime to convert it to an Android app was 526 ms (locally) and 94 ms (EC2). While the faster EC2 runtime

seems surprising, code generation involves copying and instantiating a template Android folder and Windows 10 is known[1] for slower file copying.

2.5.2 RQ2: More Accurate Than Teleport

To compare Doodle2App with Teleport, we used our test samples of the 7 classes that overlap with Teleport (i.e., squiggle, square, checkbox, switch, star, dropdown, and slider), yielding 712 samples. We converted each of these samples from a QuickDraw stroke sequence to an image and passed the image to Teleport's vision API. The average response time (between request and response) was some 313 ms, which was likely dominated by internet transmission delays between us and the Teleport server.

Out of the 712 test samples Teleport classified correctly 124 (17.4%). To put this low value into the context of the 85% accuracy given on Teleport's website, the same website also calls out problems with recognizing sliders and ratings (stars) due to the Teleport designers using fewer training samples on these two classes compared with their other classes. On the same set of 712 samples Doodle2App achieved an accuracy of 93.9%.

2.6 Conclusions

User interface development typically starts with freehand sketching, with pen on paper, which creates a big gap in the software development process. Recent advances in deep neural networks that have been trained on large sketch stroke sequence sample collections have enabled online sketch detection that supports many sketch element classes at high classification accuracy. This paper leveraged the recent Google Quick, Draw! dataset of 50M sketch stroke sequences to pre-train a recurrent neural network and retrained it with sketch stroke sequences we collected via Amazon Mechanical Turk. The resulting Doodle2App

[1] `https://superuser.com/questions/1124472/why-is-linux-30x-faster-than-windows-10-in-c` accessed March 2020.

website offers a drawing interface and an interactive UI preview and can convert sketches to a compilable Android application. On 712 sketch samples Doodle2App achieved higher accuracy than the state-of-the-art tool Teleport.

CHAPTER 3

PSDoodle: Fast App Screen Search via Partial Screen Doodle

3.1 Introduction

Searching through existing repositories for a specific mobile app screen design is currently either slow or tedious. Currently such searches are either limited to traditional keyword searches (e.g., via Google's image search) or require as input a complete query screen image (i.e., via the SWIRE line of work [34, 35]) and are therefore slow and do not support well an interactive or iterative search style.

Having an effective and efficient search engine for mobile app screens can benefit many key software engineering tasks, including requirements gathering, understanding current market trends, analyzing features, providing inspiration to developers, and as a benchmark for evaluation [36, 37]. Given the wide-spread and increasing use of mobile apps in a "mobile first" world and the resources spent on developing them [4, 5], such a screen search engine could have a large positive impact on many software developers and users. We are particularly focused on software developers with little to no UI/UX/design background. These users may only have a vague idea of the screen contents and are looking for inspiration from professional screen designs.

Several screen repositories exist, including websites such as Dribbble[1] and Behance[2]. Another repository is the Rico dataset curated from Android apps at runtime [16]. Searching through this vast collection and finding desired example screens currently requires extensive effort via keyword-based search (e.g., for screen color, theme, date, and location)

[1] `https://dribbble.com/`, accessed January 2022.

[2] `https://www.behance.net/`, accessed January 2022.

through several websites [38]. Moreover, novice users often fail to formulate good keyword queries and therefore do not get the intended search results [36].

Several researchers have proposed using visual, e.g., image- or sketch-based search methods because they are easy to use and fast to adopt [39]. For software development sketches are a natural fit, as sketches are a common form of visual representation, especially during early software development phases such as UI prototyping [1, 2, 18, 19, 20].

PSDoodle is the first approach that supports interactive and iterative sketch-based screen search. PSDoodle uses a digital drawing interface with support for touchscreen devices of different resolutions and provides ease of use for the mouse. Using a digital drawing interface enables live search and user interaction. PSDoodle also does not suffer from the processing delays of paper-based approaches with their offline processing steps.

Figure 3.1 gives an overview of a sample PSDoodle search, starting in row 1 with the user sketching a “hamburger”-style menu icon in the top left corner. Each of PSDoodle’s top-5 result screens contains a hamburger menu icon at about the sketched location. The second row shows the result of the user adding the doodle of a custom image below the hamburger icon, followed by two rows adding one more UI element doodle each.

PSDoodle employs deep learning to identify UI elements from drawing strokes. PS-Doodle fetches real-world UI examples from the Rico [16] dataset based on UI element type, position, and element shape. It retrieves UI screens from the first UI element query element. PSDoodle updates the search result with the addition or removal of a UI element in the sketch.

At the same time PSDoodle provides search accuracy on par with state-of-the-art full-screen sketch approaches. We compared PSDoodle to state-of-the-art approaches by recruiting and observing 10 participants who used PSDoodle for the first time. We displayed a UI screenshot from Rico and instructed the participant to draw until the Rico screen appears in PSDoodle’s top search results. 88% of the time PSDoodle retrieved and

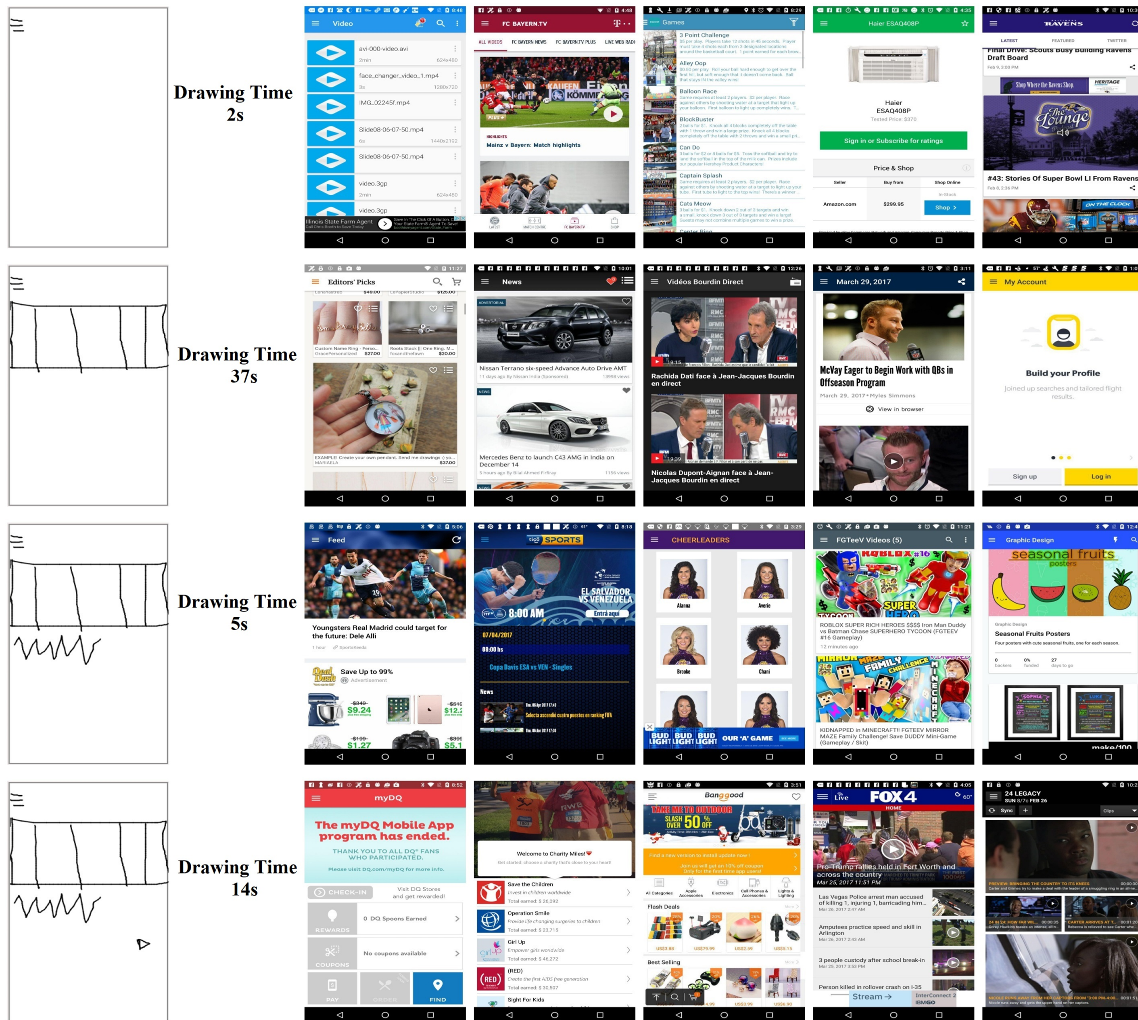


Figure 3.1. The second app screen search one of the study participants performed on PSDoodle (after finishing a 7-minute tutorial). Each row shows a user query sketch (1st column) after adding one more UI element, followed by PSDoodle's top-5 (out of 58k) search results (in order). For each of these four queries, several of the result screens contain the sketched UI elements at about the sketched location. Drawing time contains all time from the user starting to work on the new UI element to the user indicating the new UI element sketch is finished. In each row PSDoodle returned the top-10 (ranked) result screens within 2 seconds (which includes a roundtrip from the user's machine to the AWS-hosted PSDoodle)..

displayed the Rico target screen in its top-10 search results. A user usually spent an average of 107 seconds and drew an average of 5.5 elements during the process. This compared

favourably with the most closely related tool SWIRE [34], which took 246 seconds to complete a sketch and took an average of 21.1 icon elements in each query drawing. To summarize, this paper makes the following major contributions.

- PSDoodle is the first tool that provides an interactive iterative search-by-sketch screen search. It is freely available online at: `http://pixeltoapp.com/PSDoodle/`
- In our comparison with the state-of-the-art SWIRE line of work, PSDoodle achieved similar top-10 search accuracy while requiring less than 50% of the time.
- All of PSDoodle's source code, processing scripts, training data, and experimental results are available under permissive open-source licenses [40, 12].

3.2 Background

Due to their wide use, we focus on Android apps and their common UI elements. Rico contains 72k unique app screens, collected from 9.3k Android apps from 27 app categories of the Google Play store [16]. Rico ran (via ERICA [41]) each of these Android apps on modified Android classes to efficiently capture both screenshots and each screen's runtime UI view hierarchy. Rico thereby provides for each screenshot each UI element's Android class name, textual properties, x/y coordinates, and visibility.

A common challenge is understanding apps' custom UI elements (e.g., a clickable custom image used as an alternative implementation of a standard Android icon). To understand an UI element's intent beyond its Android class name, Liu et al. clustered 73k Rico screen elements by image similarity, the similarity of an element's surrounding text snippets, and similar code-based patterns [42]. This yielded 25 UI component types (e.g.: checkbox, icon, image, text, text button), 197 text button concepts (e.g.: no, login, ok), and 135 icon classes (e.g.: add, menu, share, star), with which Liu et al. labeled all screen elements in Rico.

3.2.1 SWIRE: Offline Full-screen Search

Most closely related to our work is SWIRE [34]. SWIRE collected 3.8k low-fidelity full-screen Rico screen sketches from 4 experienced UI designers given a pre-defined drawing convention. Specifically, SWIRE instructs users to sketch each image as a crossed-out square (square borders plus diagonals) or as a square filled with a mountain outline. SWIRE users also represent any text with (a part of) the same 3-word template (Lorem ipsum dolor) or by squiggly lines. SWIRE trained a deep neural network on 1.7k Rico sketch-screenshot pairs created by 3 designers, yielding a top-10 screen retrieval accuracy of 61% (i.e., in 61% of cases the screenshot corresponding to the fourth designer's query sketch was one of SWIRE's top-10 search result screenshots).

SWIRE reflects a traditional paper-based design style. Users sketch with pen on paper inside an Aruco marker frame [43] to streamline subsequent de-noising, camera angle correction, and projection correction. To change a sketch the user will likely have to start over. Even scanning or taking a snap once plus the subsequent processing steps requires significant time.

Recent SWIRE follow-up work reported a top-10 accuracy of 90.1% [35]. While using different processing steps, at a high level it followed SWIRE's paper-based design style and thus faces similar challenges for interactive search.

3.2.2 Google QuickDraw & DoodleUINet

Google's Quick, Draw! ("QuickDraw") offers some 50M doodles of 345 everyday categories, from "aircraft carrier" to "zigzag" [10, 11]. QuickDraw doodles were sketched by anonymous website visitors, who were only given a one-word description of the thing to sketch. For each element category this yielded sketches performed in a wide variety of drawing styles. Given this diverse training set, QuickDraw achieved solid doodle recognition accuracy for a wide range of sketching styles (e.g., earlier work reported some 70%

top-1 doodle recognition accuracy [44]). Internally QuickDraw represents each doodle as a stroke sequence. Each stroke is a drawing from a start-touch to an end-touch event (e.g., mouse button press and un-press), represented by a sequence of straight lines.

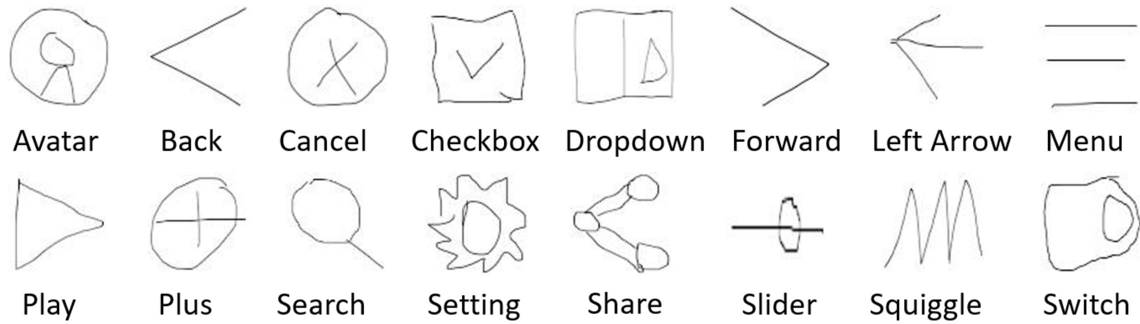


Figure 3.2. PSDoodle's DoodleUINet icons (1 sample per class)..

DoodleUINet offers some 11k crowdworker-created doodles of 16 common Android UI element categories [12]. Figure 3.2 visualizes DoodleUINet's 16 UI element categories, spanning Android built-in element types (e.g., checkbox) and custom-designed images (e.g., avatar). DoodleUINet doodles are stored in QuickDraw's format but do not overlap with QuickDraw's doodle categories. In contrast to QuickDraw's flexible doodle recognition, the current version of DoodleUINet focuses on a *single drawing style* per element category ("stylized"), which it achieved by briefly presenting crowdworkers a stylized target image of the element they should sketch. (For example, in DoodleUINet a UI element to reach "settings" currently always looks like a gear symbol.) Some 10k DoodleUINet sketches are labeled "correct" (or similar-looking to the target image according to manual review) and some 1k are labeled "incorrect".

3.3 Overview and Design

Figure 3.3 gives an overview of PSDoodle's architecture. PSDoodle offers a drawing interface (bottom left) and recognizes a stroke sequence as an UI element via a deep neural network trained on DoodleUINet and QuickDraw doodles. After recognizing a new UI element, PSDoodle looks up the top-N matching screens in its dictionary of Rico screen hierarchies via PSDoodle's similarity metric based on UI element shape, position, and occurrence frequency.

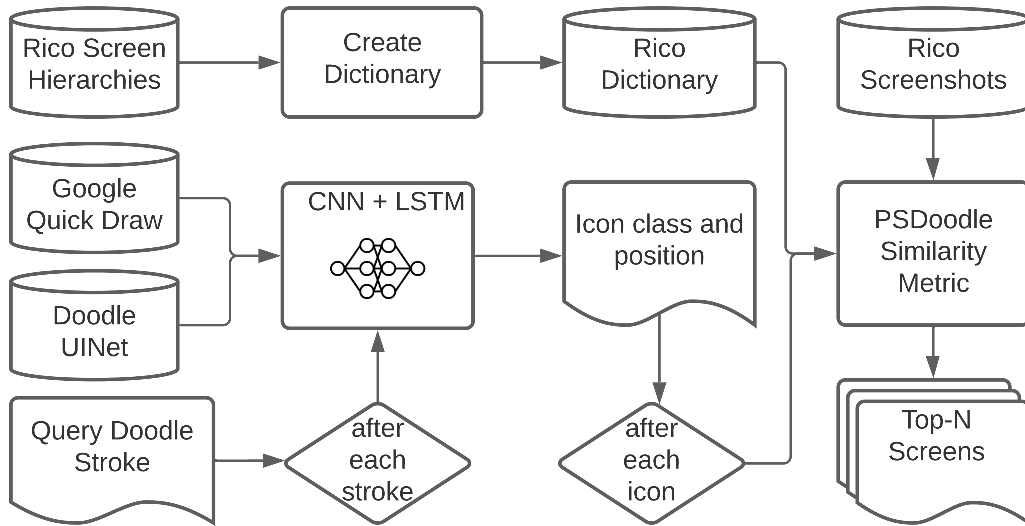


Figure 3.3. PSDoodle answers user queries via an offline-trained icon-level stroke-sequence recognizer and an offline-created hierarchy dictionary of 58k Rico screens..

3.3.1 Rico Screens & UI Element Labels

While the Rico paper mentions 72k screens, its dataset contains 66,261 screens. Given our UI element based search, PSDoodle cannot distinguish between screens with few UI elements (e.g., between two screens that only show a single large image). We thus exclude a Rico screen if the entire screen consists of a single text area (2,384 screens), a

single image (561), single text plus single image (502), single webview (2,367), webview covering most of the screen area (1,433), or has no hierarchy information (888). (While a webview may contain an arbitrary webpage, Rico contains no information about this webpage.) This yields 58,126 Rico screens in PSDoodle.

Container's Android Class	**Element's Android Class**	**New Label**
CheckedTextView, AppCompatCheckedTextView, AppCompatCheckBox, CheckableImageView, AnimationCheckBox, CheckableImageButton, CheckBox, ColorableCheckBoxPreference	AppCompatCheckBox, PreferenceCheckbox, CheckboxChoice, CheckboxTextView, CenteredCheckBox, StyledCheckBox, CheckButton, AppCompatCheckBox, CheckBox, CheckBoxMaterial	Checkbox
RangeSeekBar, SeekBar	RangeSeekBar, TwoThumbSeekBar, EqSeekBar, PriceRangeSeekBar, VideoSliceSeekBar, SliderButton	Slider
RatingBar	RatingWidget, RatingSliderView, RatingView, Rating	Star
SwitchCompat, Switch	SwitchCompat, CustomThemeSwitchButton, BetterSwitch, LabeledSwitch, CustomToggleSwitch, CheckSwitchButton, CustomSwitch, MySwitch, SwitchButton, Switch	Switch
n/a	CustomSearchView, SearchEditText, CustomSearchView, SearchBoxButton	Search

Table 3.1. 47 patterns of fixes we have applied to "input" and "image" labels Liu et al. have applied to Rico UI elements on 3,317 screens. If Liu et al. labeled an element "input" or "image" and the element's type is a second column Android class or its direct container is a first column Android class, then we replace the label with the right column.

For 3,317 Rico screens we noticed and fixed several inaccuracies in Liu et al.'s labeling of UI elements as "input" or "image". Table 3.1 summarizes these fixes as 47 patterns. For example, we found that if Liu et al. labeled a UI element of Android class AppCompatCheckBox as an "input" then that UI element really looks like a "checkbox".

3.3.2 Query Language: Stylized + Flexible Doodles

While our long-term goal is to support every user and their preferred query styles (i.e., via an arbitrary mix of individual sketching styles, keywords, and structured query languages), PSDoodle focuses on sketch-only screen search. Specifically, PSDoodle combines the stylized DoodleUINet sketch style with the flexible QuickDraw sketch style. QuickDraw has already validated that supporting both many categories and flexible drawing styles is possible using the QuickDraw representation and classification PSDoodle has adapted. So migrating PSDoodle to support more UI element categories and a flexible drawing style is mostly a matter of collecting more training samples (and retraining PSDoodle's neural net).

A key challenge not addressed by QuickDraw is sketching deeply nested composite structures, which is common in app screens (e.g., a list of images plus text pairs in a container that is just one part of the screen). PSDoodle supports sketching such screens via its sequence-of-elements style. Specifically, the PSDoodle doodle classifier recognizes one UI element at a time. So once a user starts sketching a new doodle, PSDoodle treats each stroke as belonging to that UI element doodle, until the user indicates the UI element sketch is done. In that style it does not matter if the user first sketches a container or one of its (nested) UI elements, PSDoodle recognizes each separately and treats them as separate elements, allowing arbitrarily deeply nested container structures.

Figure 3.4 shows how PSDoodle presents its query language to its users (as a "cheatsheet"). At the individual UI element level, DoodleUINet [12] is a good fit for Android

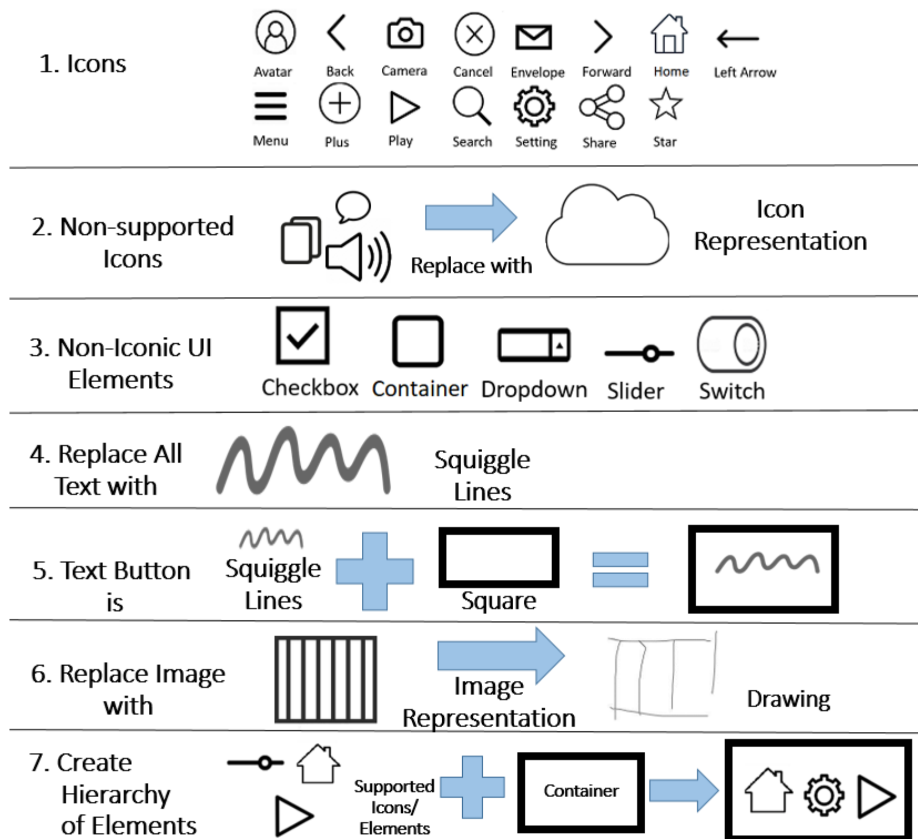


Figure 3.4. PSDoodle's query language, as presented to users..

screen search as (according to the number of element labels inferred by Liu et al. [42]) DoodleUINet covers several of the most popular UI elements in Rico. Specifically, 11/16 of the stylized DoodleUINet doodles look like the corresponding UI elements grouped and labeled by Liu et al. The other five either match SWIRE's language (squiggly line) or appear to be reasonable representations of common app concepts (dropdown, left arrow, slider, and switch). In addition to DoodleUINet, we reviewed the QuickDraw categories, looking for doodles that could be used to cover additional UI elements. We thereby identified 7 QuickDraw classes (Figure 3.5 shows one sample each) that in our subjective judgement were a good match for UI sketching.

Besides the relatively close doodle-to-screen similarity of several classes (e.g., a sketched “menu” icon looks quite similar to an on-screen menu icon), PSDoodle follows SWIRE’s approach of using a few placeholder elements to represent text and arbitrary images. For text PSDoodle uses a squiggly line (as SWIRE) and for an arbitrary image we use QuickDraw’s “jailwindow”. Furthermore, PSDoodle uses QuickDraw’s cloud to represent a default (otherwise not directly-supported) icon and QuickDraw’s square as a container.

Taken together, PSDoodle thereby covers the most common UI elements in Rico (in order) as follows (bold is from DoodleUINet, italic from QuickDraw): *Container*, *image*, *icon* (a small interactive image), **text**, ***text button***, web view, input, list item, **switch** (a toggle element), map view, **slider**, and **checkbox**. Rico further sub-categorized the most popular icon types (#3 in the above list) as **back**, followed by (in order) **menu**, **cancel** (close), **search**, **plus** (add), **avatar** (user head-shot type image), *home*, **share**, **setting**, *star*, edit, more, refresh, **forward**, and **play**. PSDoodle further supports *camera*, **dropdown**, *envelope*, and **left arrow**.

Some popular UI elements can be treated as compound elements that can be composed of other more basic ones. PSDoodle supports one such case, i.e., the Android text button as “text” inside a “square”. If a squiggle is inside a square and the square has no other nested UI elements then PSDoodle merges these two elements into a single (compound) element.

3.3.3 UI Element Doodle Recognition

Users draw on PSDoodle’s website via mouse or touch events. Users can undo or redo strokes and remove the last element doodle (Figure 3.6 top left). Each time the user adds a stroke to the current doodle, PSDoodle shows its current top-3 UI element predictions (top right). A user can pick any of these three (and tap “Icon done”) or continue editing the current UI element doodle.

Once the user taps “Icon done”, PSDoodle adds the sketched UI element to its search query, issues the query, and updates the display of its top-N Android search result screens.

To recognize a single UI element from strokes, we trained a deep neural network using QuickDraw’s network architecture [44], i.e., a 1-D convolutional neural network (CNN) layer (48 filters, kernel size 5) followed by a 1-D CNN layer (kernel size 5, 64 filters), a 1-D CNN layer (kernel size 3, 96 filters), 3 Bi-LSTM layers, and a fully-connected layer.

We used DoodleUINet (some 600 doodles labeled “correct” for each of the 16 classes) plus a random 600-doodle sample of each of our 7 QuickDraw classes.

We used transfer learning [32] to pre-train the CNN layers for 23 QuickDraw classes outside our 7 QuickDraw classes. We then split our 23 classes into training and test samples (80%/20%) and trained the network for 24,893 steps, which yielded an accuracy of 94.5% on the test data (which is similar to the 94.2% accuracy a recent study achieved with 7-class QuickDraw subset [9]).

To map an input stroke to QuickDraw’s stroke-5 format [10], PSDoodle normalizes input stroke int locations to floats. Specifically, in the stroke-5 format [10] each user input stroke is a sequence of points where each point is a tuple ($\Delta x, \Delta$ y, p1, p2, p3). Here p1 to p3 are binary sketch states after the current vertex (touching the canvas, raised from the canvas, done). Δx and Δy are integer pixel distances we normalize to floats (maintaining, among others, the number of vertices between input and normalized image).

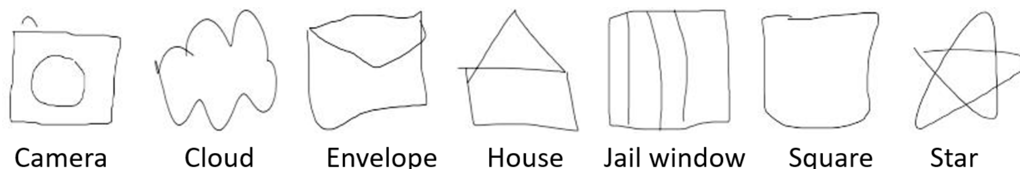


Figure 3.5. PSDoodle’s QuickDraw icons (one sample per class)..

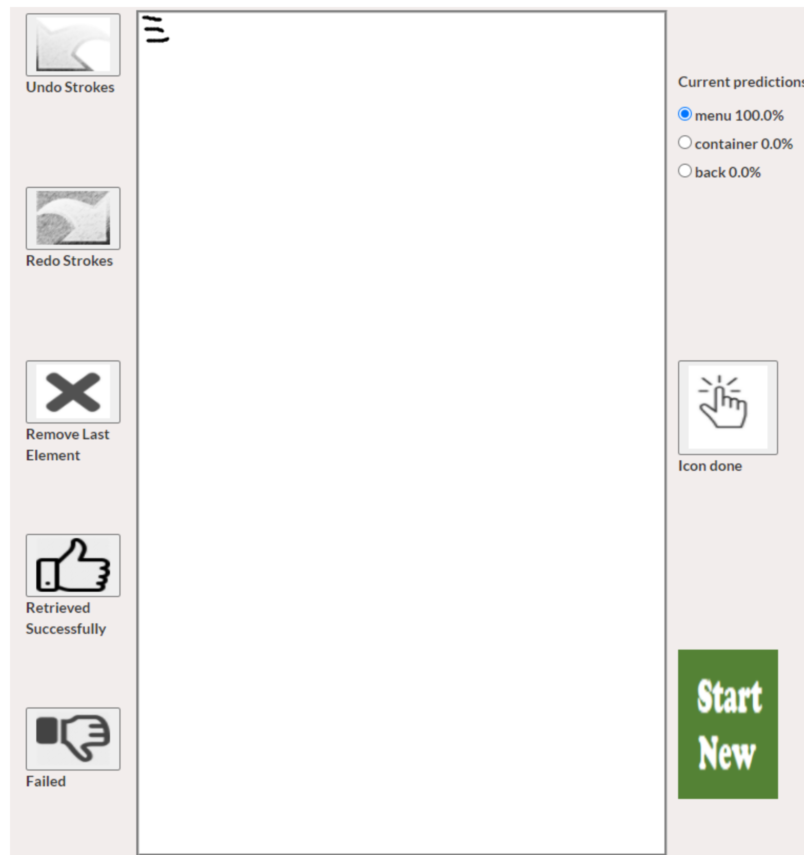


Figure 3.6. PSDoodle drawing UI, under which PSDoodle shows its current top-N Android search result screens (omitted)..

3.3.4 Searching Screens for UI Element Doodles

After the user adds (or removes) a UI element, PSDoodle displays Rico screens that are similar to the current (partial) user screen sketch. PSDoodle scores each of the 58k Rico screens based on how closely the screen matches the query doodles' presence, position, and shape. A key challenge is that a sketch is an abstract representation that is geometrically relatively far apart from its real-world counterpart. A UI element doodle thus will likely not be in the exact scale and position as it should appear on a UI screen. A similarity metric based on exact matching is thus likely to fail in sketch-based screen search.

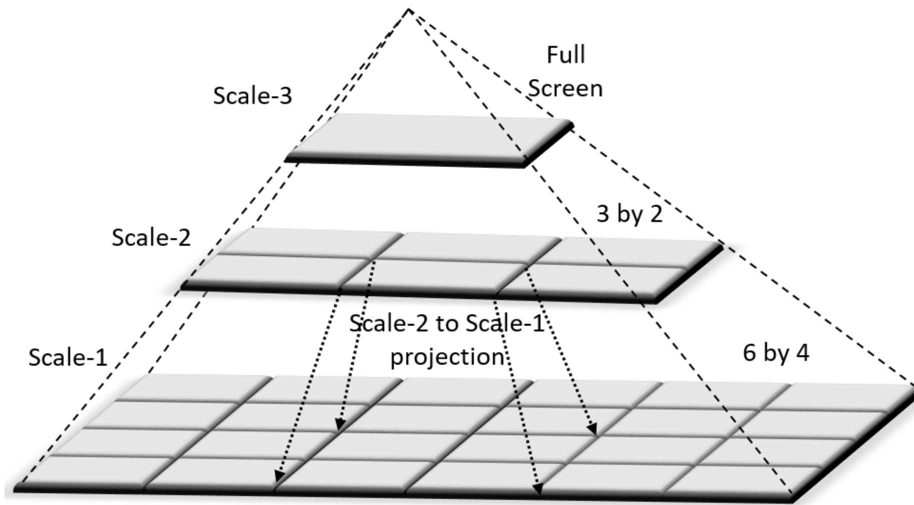


Figure 3.7. PSDoodle's 3 levels of UI element search granularity..

To address the issue, PSDoodle matches doodles (as recognized by the neural network after merging compound elements) with screen elements at different levels of screen resolution, starting at a fine-grained level but then backing up to more coarse-grained matches. Specifically, PSDoodle's fine granularity scale-1 divides the canvas into 24 equal-sized rectangles (6 rows of 4 tiles each), scale-2 groups these into 6 rectangles (3 rows of 2 tiles), and finally scale-3 is a single rectangle. Figure 3.7 gives an overview of the different scale levels and how moving to a higher level widens the search area for a UI element match.

Besides PSDoodle's 3-level matching of a UI element's screen location, PSDoodle also matches the number of UI elements of a given class and takes into account how rare a UI element is within all screens. PSDoodle thus boosts the score of a rarer UI element as its presence may be more significant to the user. PSDoodle computes the inverse document frequency (IDF) of each UI element type in the 58k Rico screens (where a UI element type's IDF is larger if it appears on fewer screens).

For fast screen retrieval, PSDoodle maintains a dictionary of Rico's 58k ("original") screens. This dictionary maps each of PSDoodle's 24 UI element types to a list of screens,

Algorithm 1 scores screens by how closely they match the query doodles' size and location; p_1, p_2, p_3, Δ_w, and C_w are hyperparameters; Δ_A and Δ_C are UI element area- and count-differences $[0..1]$.

```
1:  res ← {}                                                    // Score per original screen
2:  for doodles in sketch do                                    // Doodles of same type
3:      for screen in dict[class(doodles)] do                   // Screen + tiles
4:          z ← p_3                                             // Screen's score
5:          for tile in tiles(screen) do                        // Scale-1 tile
6:              (A_o, C_o) ← elemAreaAndCount(screen[tile])
7:              if tile in tiles(doodles) then                  // Overlaps doodles
8:                  (A_d, C_d) ← elemAreaAndCount(doodles[tile])
9:                  Δ_A ← 1 − |A_d − A_o|                       // Tile area: doodle vs orig
10:                 Δ_C ← max(0, 1 − C_w × |C_d − C_o|)         // Counts
11:                 z ← z + (p_1 × A_o/C_o × A_d/C_d) + (Δ_w × A_d × Δ_A × Δ_C)
12:             else if tile in neighbor(tiles(doodles)) then
13:                 z ← z + (p_2 × A_o/C_o)
14:             end if
15:         end for
16:         res[screen] ← res[screen] + (z × idf[class(doodles)])
17:     end for
18: end for
19: sort(res)                                                   // Retrieved screens by score
```

where each screen lists for each of its tiles the percentage of the tile's area (A_o) being covered by how many (C_o) instances of that UI element type. Algorithm 1 summarizes PSDoodle's screen similarity scoring as pseudo code. The algorithm iterates through the query sketch's UI element doodles, one element class at a time (e.g., starting with all of the query sketch's squiggly line doodles taken as a group). For each element doodle group, the algorithm looks up the Rico screens that contain at least one instance of the doodle group's element type (Line 3). For each matching Rico screen, we then iterate over the screen's scale-1 tiles that contain the given doodle type (Line 5) .

For each such original Rico screen tile that contains the doodled UI element type there are now three cases. First, if the original screen tile matches a tile of the doodle group then we have a scale-1 match and compute the percentage of that tile's area (A_d) being covered by how many (C_d) doodles of that UI element type (Line 8). Then we compute the difference in the tile's area coverage percentages between doodles and original screen elements (Line 9) and the difference in how many doodles vs how many original screen elements are in that tile (Line 10). We add the resulting tile score to its screen's overall score (Line 11).

In the second case (Line 12), the screen tile containing a UI element of the doodle group's class does not overlap with any tile of the doodle group. In this case PSDoodle backs up to its scale-2 search and checks if this screen tile overlaps with any direct neighbor tiles of the doodle group's tiles. If this is the case the screen gets a smaller score boost. Finally, if there is neither a scale-1 nor a scale-2 match, then the screen's score remains unchanged.

3.3.5 Hyperparameter Optimization

Algorithm 1 mentions five hyperparameters, three for the scale levels (p_1, p_2, p_3) plus Δ_w for the weight difference and C_w for the occurrence difference. To find optimal values

for these hyper-parameters we collected 30 sketches from 5 computer science graduate students.

The collected sketches represent 30 different Rico screens that have at least two PSDoodle-supported icons. None of these screenshots or sketches were used for the tool evaluation. An exhaustive search with GridSearchCV of scikit-learn [45] to get a high score and top-rank for the target screen yielded the optimized values $p_1 = 39$, $p_2 = 8$, $p_3 = 9$, $\Delta_w = 0.4$, and $C_w = 11$.

A closer look at the hyper-parameters indicates that they give more weights to scale-1 matches compared to the other two scales. With 24 girds, a scale-1 match implies higher location similarity, which we intuitively expect to yield a higher screen score.

Table 3.2. Strokes per doodle in PSDoodle's data sets (left) and its partial doodle recognition trained on 80% classifying the other 20% (the test doodles) (right): 1st stroke at which PSDoodle ranks a doodle's correct class first (top-1) and within the top-3; W = test doodles PSDoodle classifies wrongly at the last/all strokes; W* = after retraining from scratch adding samples that remove the outer boundary of avatar, cancel, checkbox, plus; m = median; l = min; h = max; SD = standard deviation; cnt = count.

Category	**Strokes in 100%**					**20%**	**1st stroke to top-1**					**W** [%]		**W*** [%]		**1st stroke to top-3**					**W** [%]		**W*** [%]	
	avg	m	l	h	SD	cnt	avg	m	l	h	SD	lst	all	lst	all	avg	m	l	h	SD	lst	all	lst	all
Camera	3.5	3	1	9	1.1	143	2.3	2	1	5	0.7	6	4	13	8	1.8	2	1	5	0.8	1	1	1	1
Cloud	1.4	1	1	24	1.4	154	1.1	1	1	3	0.4	12	10	4	3	1.1	1	1	3	0.4	3	2	1	1
Envelope	2.1	2	1	9	1.1	145	1.9	2	1	9	1.1	7	6	6	6	1.3	1	1	4	0.6	0	0	0	0
House	3.5	3	1	23	2.2	135	2.1	2	1	5	0.9	11	9	1	1	2.0	2	1	10	1.1	3	2	1	1
Jail-win	5.8	5	2	20	1.7	143	3.4	3	2	8	1.1	7	6	1	1	2.9	3	1	8	1.2	2	1	1	0
Square	1.3	1	1	4	0.7	147	1.1	1	1	3	0.3	2	2	2	1	1.1	1	1	3	0.3	1	1	1	1
Star	1.4	1	1	10	1.0	148	1.1	1	1	4	0.5	1	1	1	1	1.1	1	1	2	0.3	0	0	0	0
Avatar	3.8	4	1	8	0.8	136	2.9	3	1	7	0.9	9	8	2	2	2.3	2	1	6	0.7	1	1	0	0
Back	1.1	1	1	12	0.7	122	1.0	1	1	2	0.2	3	2	5	3	1.0	1	1	2	0.1	0	0	4	3
Cancel	3.1	3	2	12	0.5	127	2.6	3	1	5	0.6	25	21	3	2	2.2	2	1	4	0.6	2	1	1	1
Checkbox	2.8	2	1	51	2.6	134	2.3	2	1	10	1.4	13	13	4	1	1.7	1	1	7	1.1	3	2	1	1
Drop-dwn	4.5	3	2	43	3.3	133	2.8	2	2	6	1.0	3	2	5	3	1.9	2	1	5	0.7	2	2	2	2
Forward	1.1	1	1	17	0.8	122	1.0	1	1	1	0.0	0	0	2	2	1.0	1	1	1	0.0	0	0	1	0
Left arrow	2.2	2	1	10	0.8	123	2.0	2	1	4	0.5	13	12	3	2	1.9	2	1	4	0.6	5	5	1	1
Menu	3.2	3	2	16	1.1	126	2.0	2	1	3	0.4	0	0	3	2	1.8	2	1	3	0.4	0	0	0	0
Play	1.7	1	1	15	1.2	127	1.4	1	1	2	0.5	5	4	9	3	1.3	1	1	3	0.5	2	1	3	1
Plus	3.1	3	2	11	0.7	120	2.2	2	1	6	0.9	8	8	12	8	1.8	2	1	4	0.8	2	2	3	3
Search	2.3	2	1	13	1.0	122	2.0	2	1	3	0.3	2	2	9	7	1.9	2	1	3	0.4	1	1	3	2
Setting	5.6	2	1	61	7.1	111	3.1	2	1	34	4.0	11	7	5	5	2.5	2	1	24	2.4	1	0	1	1
Share	7.0	7	1	23	1.1	117	3.7	3	2	13	1.3	3	1	5	3	2.8	3	2	7	0.8	1	1	1	0
Slider	2.6	3	1	19	1.0	134	1.9	2	1	4	0.8	4	4	2	2	1.6	2	1	4	0.6	1	1	1	1
Squiggle	1.3	1	1	52	2.5	144	1.1	1	1	4	0.4	3	1	0	0	1.0	1	1	4	0.3	1	1	0	0
Switch	3.1	2	1	16	1.6	137	2.3	2	1	6	1.0	6	4	6	5	1.6	1	1	5	0.8	0	0	1	1

3.4 Evaluation

We evaluated PSDoodle's recognition accuracy of partial UI element doodles, its top-10 retrieval accuracy of partial screen sketches, and its screen retrieval time using the following research questions.

RQ1 Can PSDoodle recognize partial UI element sketches?

RQ2 Can PSDoodle achieve similar top-10 accuracy as state-of-the-art screen search approaches?

RQ3 At a similar accuracy level, how many UI elements did participants sketch in PSDoodle compared to state-of-the-art complete-screen sketch approaches?

RQ4 At a similar accuracy level, can PSDoodle retrieve screens faster than state-of-the-art approaches?

Following the most closely related work [34, 35], we evaluated screen search performance by measuring top-k (screen) retrieval accuracy. We thus showed a participant a target screen to sketch and measured where in the result ranking the target screen appears. Top-k retrieval accuracy is the most common metric for sketch-based image retrieval tasks and correlates with user satisfaction [46].

Specifically, we evaluated screen search (in RQ2, RQ3, and RQ4) with 30 "target" Rico screens, which we selected as follows. To ensure the target Rico screens contain at least some UI elements PSDoodle supports, we removed from Rico's 58k screens those that contain less than two PSDoodle-supported UI elements, yielding 50,113 screens. From these 50k we randomly picked 30 screens, of which 26 were also in the SWIRE dataset.

We recruited 10 Computer Science students (all ages under 30). None of the participants had any formal UI/UX design training. All participants had heard about mobile app development principles before the study. For diversity, we recruited 5 participants (1 female, 4 male) without plus 5 (2 female, 3 male) with some prior mobile app development

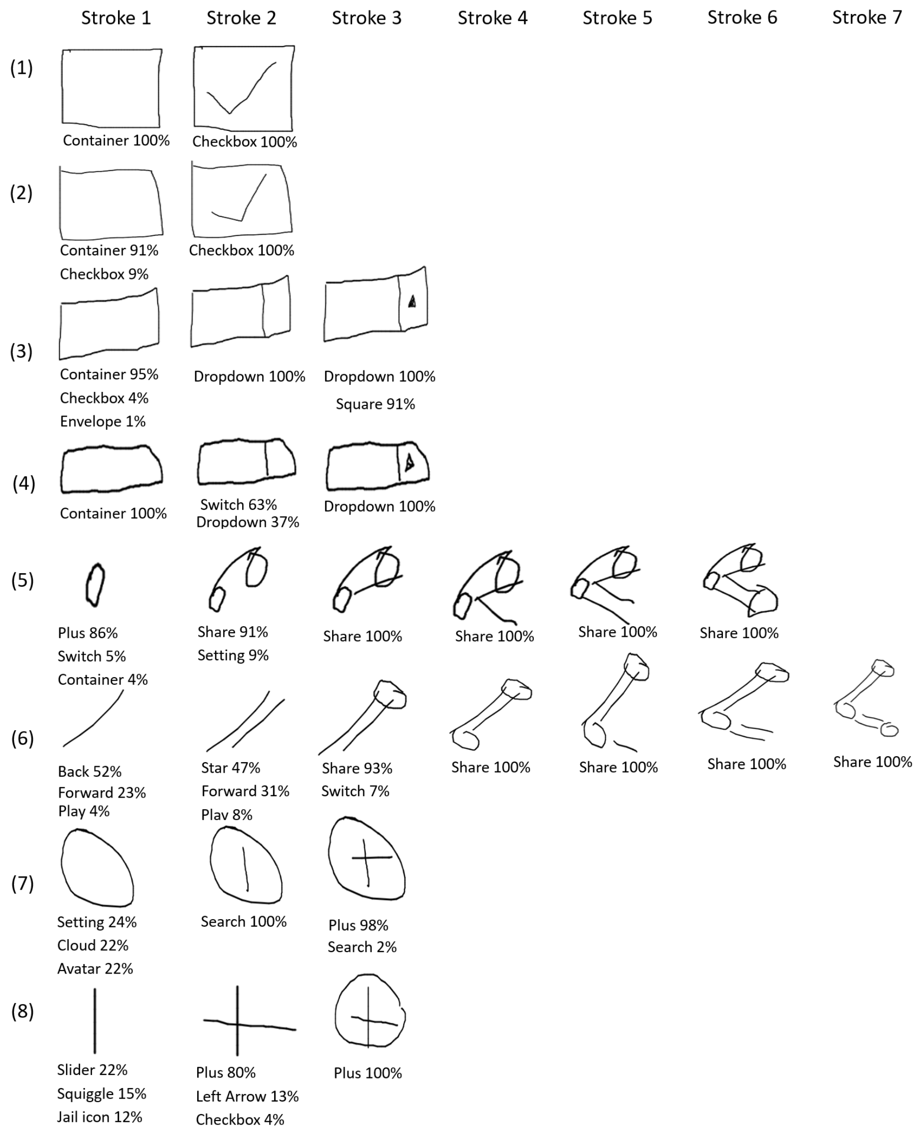


Figure 3.8. Two random samples (from the 20% of doodles) from 4 categories. Below each incremental stroke is PSDoodle's confidence for its current top predictions. For 6 of these 8 doodles PSDoodle reached the correct prediction before the last stroke, allowing the user to communicate their intent without finishing the doodle..

experience. Each student was compensated with USD 10 and used PSDoodle for the first time.

For the experiment, we used PSDoodle's regular setup as a website hosted on an Amazon AWS EC2 general purpose instance (t2.large) with two virtual CPUs and 8 GB

of RAM. Each participant interacted with PSDoodle over the internet from their personal machine (i.e., a laptop or desktop computer). Each participant first spent an average of 9 minutes on the PSDoodle's interactive tutorial, which covers PSDoodle's visual language, how and where to draw, how to access the cheat sheet (Figure 3.4), how to see the search results, and when to stop the search (`http://pixeltoapp.com/toolIns/`).

After the tutorial each user was instructed to sketch at least 3 screens. The PSDoodle website recorded their drawings, drawing time, and query results. Throughout the experiments, we observed participants' performance via screen sharing but did not otherwise interact with them (e.g., to coach them on how to use PSDoodle). All records are available in the PSDoodle repository.

3.4.1 RQ1: Recognizing Partial Icon Doodles

Table 3.2 compares the number of per-doodle strokes in PSDoodle's data sets with the number of strokes PSDoodle takes to correctly classify a doodle. For the latter Table 3.2 lists two criteria, ranking the correct class first (middle columns) and ranking the correct class in the top-3 (right columns). In the experiments participants often kept sketching until PSDoodle ranked the correct class top-1 but sometimes stopped sketching after selecting the correct class from the top-3 predictions the PSDoodle UI provides after each stroke.

For both criteria (top-1 and top-3), users can often transmit their intent to PSDoodle with fewer than a doodle's full set of strokes. For example, while the average avatar doodle contains 3.8 strokes, PSDoodle ranks avatar top-1 on average after 2.9 strokes and top-3 after just 2.3 strokes. Several other classes have similarly large reductions in average stroke counts.

To visualize PSDoodle performance on concrete examples, Figure 3.8 displays PSDoodle's current prediction after each stroke of 8 randomly sampled drawings from 4 categories. For example, in the fifth row PSDoodle ranks the doodle's correct class top-1 after

only two strokes of the doodle's six total strokes. Such an early correct classification allows the user to quickly move on to the next doodle, thereby saving time and receiving query results faster.

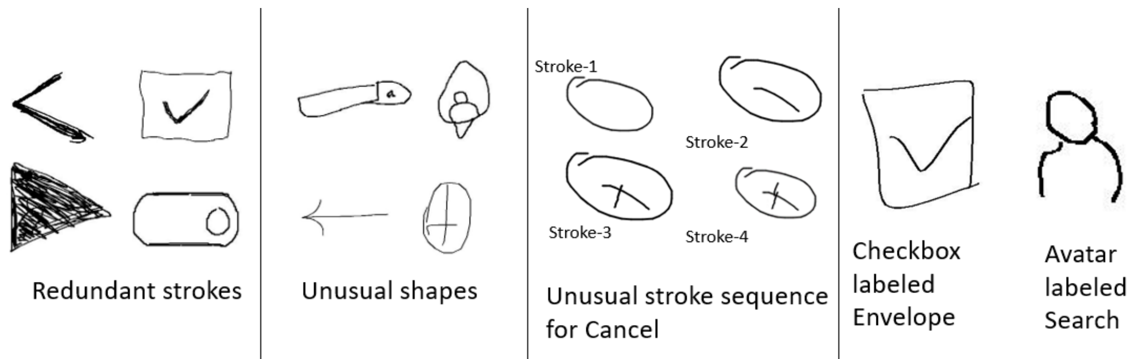


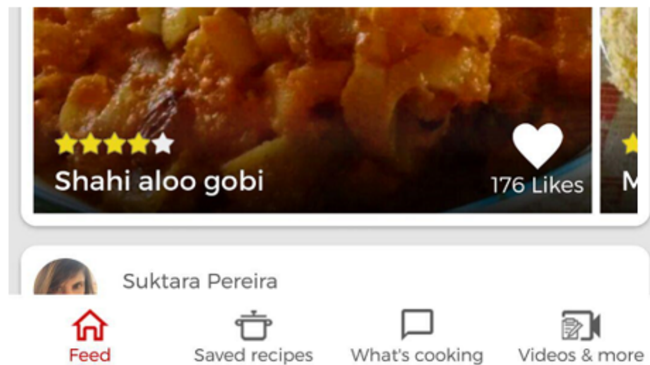
Figure 3.9. Test set drawings the network misclassified..

We also inspected a random sample of 35 of the 192 test set sketches PSDoodle misclassified after the last stroke. Among these 35 samples we found four patterns, i.e., being an outlier due to using a large number of strokes (compared to the doodle class's average) for 7/35 samples, using unusual strokes (such as a squiggle during drawing using more vertices compared to the class's vertex average) or stroke sequence for 10/35 samples, deviation from the class's common shape (14/35), and resembling another category (4/35). Figure 3.9 shows examples of these four categories.

3.4.2 RQ2: Top-10 Screen Search Accuracy

For this experiment participants were instructed to sketch with the goal of using PSDoodle to retrieve a given Rico "target" screen. We then measured how quickly this target Rico screen appeared in PSDoodle's top-10 search results. We asked participants to use the tool at least 3 times, with 4 users attempting one additional sketch each, yielding 34 screen sketches of 30 Rico screens. For these 34 sketches, 30 times the target UI screen

appears in the top-10 results, yielding a top-10 accuracy of 88.2%. (Since PSDoodle shows rows of result screens similar to Google's image search, top-1 accuracy is less relevant.) PSDoodle's top-10 accuracy is significantly higher than SWIRE's and remains similar to a recent SWIRE follow-up work by Sain et. al [35], which reported 90.1% top-10 accuracy for SWIRE sketches.



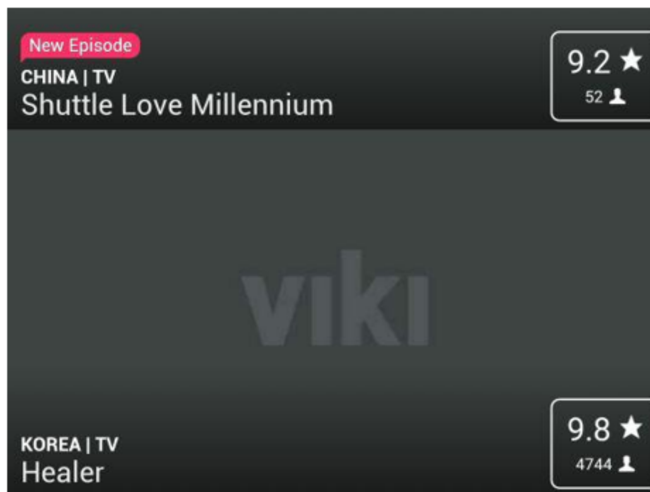


Figure 3.10. PSDoodle fails to rank these two target screens (excerpted) in its top-10 search results, due to their Rico hierarchy having no information about an icon drawn by a user..

We manually checked each case where PSDoodle failed to rank the target screen in the top-10. Figure 3.10 shows excerpts of two such screens. In both the Rico hierarchy does not contain the correct label of a user-drawn icon. Such cases could be reduced by further improving Rico's UI element clustering and classification. Another case stems from human error (i.e., participant 9 in Table 3.3) because the user selected the wrong doodle category from PSDoodle's top-3 prediction. Such human errors may become less common once users become more experienced with using PSDoodle.

3.4.3 RQ3: Search With Partial-screen Sketches

In addition to recognizing partial UI element sketches, PSDoodle also supports an iterative search style where a user refines the search results one UI element at a time. As the SWIRE-style approaches process complete-screen sketches, this research question quantifies the difference in UI elements a user has to draw to perform a successful screen search. Answering this question is made easier by the earlier experiment yielding a similar top-10 accuracy for PSDoodle and the most-accurate SWIRE-style approach.

In the 30 target Rico screens participants used, the average UI element count was 21.1 with a median of 19 (low 14, high 35, and standard deviation 5.4). SWIRE instructs users to sketch all screen elements, so a SWIRE sketch has a similar number of UI elements. In the participants' 34 PSDoodle sketches of these 30 screens the average UI element count was significantly lower at 5.5 with a median of 5 (low 3, high 9, and standard deviation 1.8).

We also tested how the most-accurate SWIRE-style approach would perform on the partial screen sketches our participants produced with PSDoodle, by training their network to the reported 90.1% top-10 and 67% top-1 accuracy for SWIRE sketches [35]. We thus converted our 34 participant sketches from QuickDraw's sequence-of-stroke format to SWIRE-style black/white bitmaps (and included them in the PSDoodle repository). We

removed the 8 participant sketches that used jail-window, as SWIRE uses a different placeholder for images. For all resulting 26 query sketches, the SWIRE follow-up failed to fetch the target screen in the top-10.

3.4.4 RQ4: Interactive and Fast Screen Retrieval

In our experiments we told participants to search via sketching for 3 minutes and stop sketching if the target screen appears in PSDoodle's top-10 result. We recorded how long each search session took. SWIRE reports that the sketching alone of each SWIRE sketch of a Rico screen took an average of 246 seconds.

Table 3.3. Time (seconds) a participant (P) took to iteratively sketch a target screen and retrieve result screens. The final target screen ranking (r) was top-10 accurate in 88% of cases.

P	**Target 1**		**Target 2**		**Target 3**		**Target 4**	
	t	r	t	r	t	r	t	r
1	55	2	51	1	134	8	-	-
2	86	3	259	7	206	3	-	-
3	134	3	97	4	63	2	-	-
4	85	5	75	1	64	5	-	-
5	202	3	60	1	105	14491	-	-
6	127	2	119	1	109	9	-	-
7	44	2	98	1	38	10	39	1
8	168	10	248	1	103	1	73	5
9	158	31	46	61	97	31	40	3
10	30	8	138	1	147	1	158	6

Table 3.3 lists the total time of the search and sketch session for each of the experiment's 34 sessions, together with the final rank of the target Rico screen in PSDoodle's search results. Total sketch and search times per search session varied from 30 to 259 seconds. While achieving similar top-10 accuracy, most of these session times were significantly shorter than the 246 second average of SWIRE for sketching only. Most of PSDoo-

dle's session times were also significantly shorter than the 180 seconds target provided to participants.

PSDoodle is deployed in AWS and supports interactive search. In our experiments there was less than 2s delay between the user submitting a search query (e.g., by pressing "icon done") to the update of the top-10 result screens on the user's PSDoodle website. Besides communication to and from AWS, the main time components were sketch recognition (below 0.1s) as well as screen similarity calculation and screen ranking (below 1s).

3.4.5 High-level Feedback from Participants

Two of our 10 participants opted out of our post-evaluation survey, leaving us with 8 completed surveys. In one question we asked how participants prefer to sketch. Three preferred touch to sketch on a larger device such as an iPad, two preferred sketching on paper and taking a picture with their phone, two wanted to use a mouse to sketch on a non-touch device, one wanted to touch to sketch on a smaller device such as a smartphone, and none wanted to scan a paper-based sketch. Overall participants preferred device-based over paper-based sketching by 3:1.

In another question we asked participants to choose from three sketch-based search tool options. Two participants voted for an approach that shows its search result only after finishing a complete screen (containing all the UI elements that the screen should have in the app). The other six participants preferred a search tool that shows live search results (i.e., search results that update when adding or removing a UI element). None of the participants picked the third option, a tool that only shows its results after sketching a partial screen containing several icons.

3.4.6 Relaxing PSDoodle's Query Language

In informal but more concrete feedback, participants explained how they sometimes struggled with the four PSDoodle's icon classes whose shapes include outer boundaries such as avatar's "outer ring". These classes were avatar, cancel, checkbox, and plus. While some participants preferred to sketch such icons without these outer boundaries, PSDoodle's training data sets contained only few such samples.

To address this issue, after the experiments with participants we created additional samples from PSDoodle's existing samples, by identifying and removing these outer boundaries. Table 3.2 lists the test doodles the version of PSDoodle used with participants classified wrongly (W) both after the last stroke and after each stroke. Retraining the doodle classifier after adding these samples yielded better recognition performance (W*). The new classifier performed worse on Camera and Search (and to a lesser degree on Back, Dropdown, Forward, Menu, Play, Plus, Share, and Switch).

Overall recognition accuracy improved from 94.5% to 94.9% (while keeping recognition speed the same), but is most notably 9% better for avatar (the class that study participants had the most trouble with). Among the 186 UI elements in the participants' 34 final screen sketches, the retrained network detected 18 UI elements with fewer strokes (while requiring more strokes for 8 UI elements).

3.5 Related Work

In sketch-based image retrieval (SBIR), computer vision techniques try to find the similarity in the sketch-image pair based on their features when a user draws an unpolished representation of the image. Earlier studies extract hand-engineered features (edge-map, Histogram of Oriented Gradients, Histogram of edge local orientation) [47, 48] to find the similarity between the pair. Deep Leaning achieves state-of-the-art performance in sev-

eral computer vision applications with Convolutional Neural Network [6, 7]. The success also draws researchers to employ deep neural networks for SBIR [49, 50]. Deep Neural Network(DNN) uses sketch-image pair for training two different networks(one for sketch and one for image). During the training phase, DNN encodes the image-sketch duo to low-dimensional feature vectors with a target to reduce the distance for similar pairs and maximize for non-similar pairs. For query, it encodes the sketch and then uses the nearest neighbor technique to query similar examples from the dataset.

Searching design from visual input (image, sketch) recently gaining attention due to the success of DNN and the creation of large-scale datasets. SWIRE [34] uses a deep neural network model to retrieve relevant UI examples from input sketches. VINS [51] UI image (wireframe, high-fidelity) retrieves UI screenshots from high-fidelity wire-frame design.

In a follow-up to SWIRE, sketching begins with a coarse-level representation of a real-world object, followed by more fine details. Rather than considering a sketch as a flat structure, they use the hierarchical structure to pair it with a photo. Two nodes of the deep neural network are fused to form the next hierarchy level by interacting and matching features between image and sketch pair. They calculated bounding boxes of the individual connected components of the SWIRE drawings to identify interest regions. A cross-modal co-attention part of the network attends to matching interest regions in a sketch and image pair. By leveraging the hierarchical traits and mutual attention between the interest region, they achieved state-of-the-art performance in the SWIRE dataset.

Successful integration of sketch in the software development process requires a large-scale UI dataset and utilization of the dataset in the deep learning model. While some freehand drawings of user interface elements are available [34, 52, 53], these sketches are available as “static” pixel-based images of the final sketch.

SWIRE [34] collected 3,802 offline sketches of 2,201 screens from 23 app categories of the Google Play store. While the SWIRE dataset is very valuable, it “only” contains

an offline snapshot of each final UI drawing. And drawings are not tagged with the UI element present in them. UISketch [53] introduced the first large-scale dataset of 17,979 hand-drawn sketches of 21 UI element categories collected from 967 participants. 69.38% of UISketch are digital sketches. The drawings are now publicly available in raw-pixel format with no stroke information.

3.6 Conclusions

Searching through existing repositories for a specific mobile app screen design is currently either slow or tedious. Such searches are either limited to basic keyword searches (Google Image Search) or require as input a complete query screen image (SWIRE). A promising alternative is interactive partial sketching, which is more structured than keyword search and faster than complete-screen queries. PSDoodle is the first system to allow interactive search of screens via interactive sketching. PSDoodle is built on top of a combination of the Rico repository of some 58k Android app screens, the Google QuickDraw dataset of icon-level doodles, and DoodleUINet, a curated corpus of some 10k app icon doodles collected from hundreds of individuals (mainly crowd-workers). In our evaluation with third-party software developers, PSDoodle provided similar accuracy as the state of the art from the SWIRE line of work, while cutting the average time required about in half. All of PSDoodle's source code, processing scripts, training data, and experimental results are available under permissive open-source licenses.

CHAPTER 4

PSDoodle: Searching for App Screens via Interactive Sketching

4.1 Introduction

App screen examples help software engineers to accumulate requirements, understand current trends, and motivate to develop a compelling mobile app [36, 37]. An effective mobile search tool might have a positive impact to keep up with the extensive and increasing use of mobile apps in day-to-day life [4].

Professional designers search for example screens via keywords through various websites including Google, Dribbble[1], and Behance[2] [51]. Keyword-based search tools consider stylistic features (e.g., color, style) or image metadata (e.g, date, location, shapes) but fail to consider the contents of the screen [54]. Moreover, novice users often fail to formulate good keywords and thus fail to get the desired search results [36].

Visual (e.g., image or sketch) search methods are easy, quick to adopt [39], and commonly used during early software development phases [1, 2, 18, 19]. Dependency on a complete query screen makes the iterative nature of the development process tedious (e.g., as in SWIRE [34] or VINS [51]). Long preprocessing pipelines for a pen on paper approach such as SWIRE include taking a snap, denoising, and projection correction.

PSDoodle is designed for a novice user who cannot or does not want to develop a complete screen at an early phase of software development [14]. PSDoodle allows a user to draw one UI element at a time. PSDoodle provides an interactive drawing interface with

[1] `https://dribbble.com/`, accessed January 2022.

[2] `https://www.behance.net/`, accessed January 2022.

support for mouse and touch screen devices. For user interaction, the drawing interface includes basic features such as redo, undo stroke, and remove the last icon.

With each stroke on the drawing interface, PSDoodle utilizes a deep neural network to classify the current UI element and instantly provides a top-3 classification with classification confidence scores. When a user finishes drawing the current UI element by pressing the "icon done" button or "d" on the keyboard, PSDoodle searches through 58k Rico [16] screens to fetch UI examples based on UI element type, position, and element shape as shown in Figure 4.1. PSDoodle fetches 80 screens and displays them at the bottom of the page within 2 seconds.

For evaluation, we enlisted ten software developers who had never used PSDoodle before and have no prior UI/UX design training. Participants used PSDoodle to draw a Rico screen until it appears in the top search results. PSDoodle's top-10 screen retrieval accuracy was comparable to the state-of-the-art full-screen drawing techniques but reduced the drawing time to one half [14].

To summarize, PSDoodle is the first tool that provides an interactive iterative search-by-sketch screen experience.While the technical-track paper describes PSDoodle's algorithm and evaluation [14], this paper adds details about PSDoodle's implementation, deployment, support for different sketching styles, ability to surface several relevant search result screens, and user survey results. All PSDoodle source code, processing scripts, training data, and experimental results are **open-source** under permissive licenses [40, 12]. The tool can be freely used at: **`http://pixeltoapp.com/PSDoodle/`**

4.2 Background

PSDoodle operates on Rico mobile app screens collected from 9.3k Android apps [16]. For each UI element in each screenshot, Rico includes the Android class name, textual in-

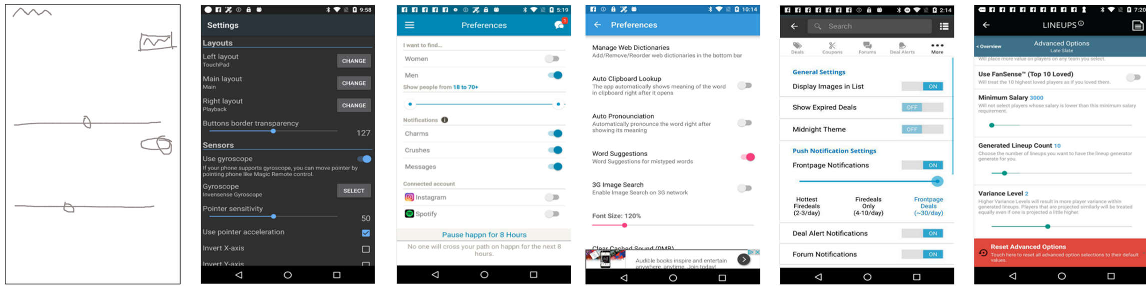


Figure 4.1. Example mobile app screen search user query (left) and PSDoodle's top five search results out of 58k app screenshots. The result screens contain all sketched UI elements at about the location they appear in the query sketch..

formation, x/y coordinates, and visibility. Liu et al. collected 73k Rico screen elements and classified all screen components in Rico into 25 UI component kinds (e.g., checkbox, icon, image, text, text button), 197 text button ideas (e.g., no, login, ok), and 135 icon classes (e.g., add, menu, share, star) [42]. PSDoodle supports several frequent Android UI elements reported by Liu et al. PSDoodle utilizes all the hierarchy information and clustering information to find similarity with a user drawing and shows Rico screens as a search result.

SWIRE [34] is the most closely related to our work. SWIRE consists of 3.8k scanned low-fidelity pen on paper drawings, each mimicking a complete Rico app screen. All drawings consist of pre-defined conventions with placeholders for image and text (e.g., square borders plus diagonals for an image). We adopted the placeholder techniques in PSDoodle for image, text, and non-supported icons. SWIRE's deep neural network achieves a top-10 screen retrieval accuracy of 61%. SWIRE's follow-up work reported a top-10 accuracy of 90.1% [35]. To generate search results, a user has to draw within a marker, take a snap of the sketch, and provide it to a long pipeline (e.g., de-noising, camera angle correction, and projection correction). For any change in the sketch, a user will most likely have to create a new complete screen sketch from scratch, scan it, and feed it to the processing stages.

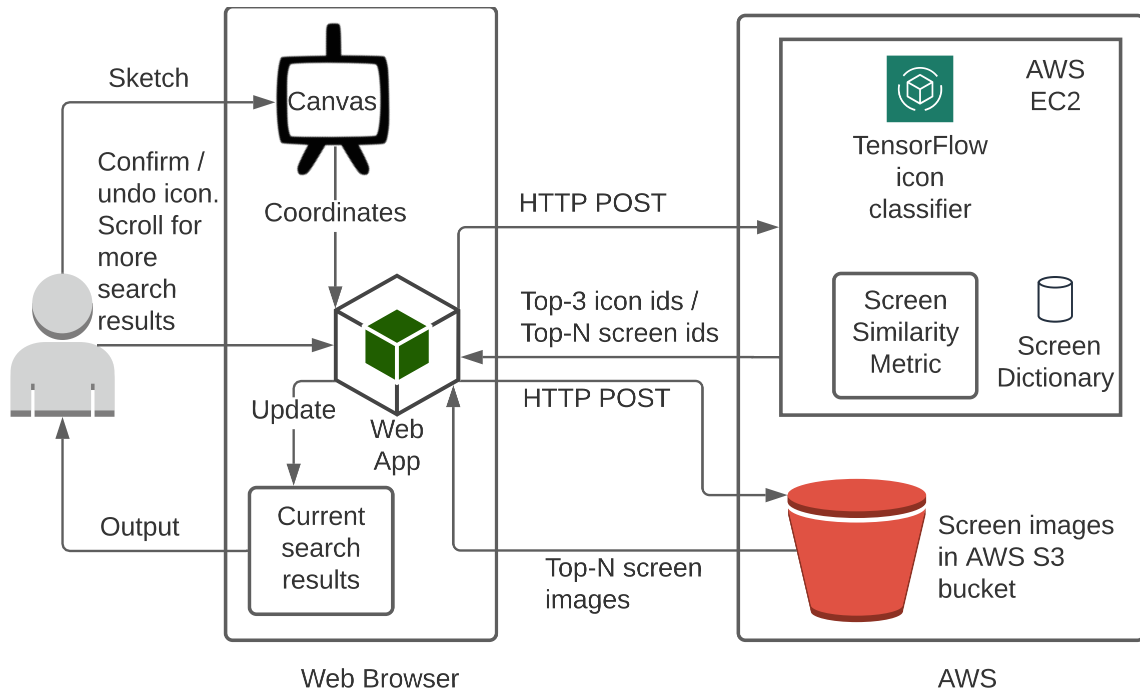


Figure 4.2. PSDoodle architecture: The user sketches on the PSDoodle webpage (from `http://pixeltoapp.com/PSDoodle`). The webpage communicates with the PSDoodle back-end hosted in AWS..

PSDoodle incorporates datasets from Google's Quick, Draw [11] ("QuickDraw") and DoodleUINet [12] to train a deep neural network for recognizing a sketched UI element. QuickDraw offers some 50M doodles of 345 everyday categories, from "aircraft carrier" to "zigzag". DoodleUINet gives 11k crowdworker-created doodles of 16 common Android UI element categories. QuickDraw and DoodleUINet represent each doodle as a stroke sequence. Each stroke consists of a series of straight lines drawn from the start to the end of a touch event (e.g., mouse button press and un-press) and each straight line is represented by the x/y coordinates of the endpoints. As an example of how such UI element doodles look like, the left screen in Figure 4.1 contains six UI element doodles—two sliders, a switch, a square, and two squiggles.

4.3 Overview and Design

Figure 4.2 shows PSDoodle's overall architecture. PSDoodle's website provides a canvas for drawing. The canvas collects the stroke information (x/y coordinates) from a user drawing and the website sends this information to Amazon AWS. PSDoodle uses a deep neural network trained using QuickDraw's network architecture [44, 14]. From the stroke information, a Tensorflow implementation of PSDoodle's neural network hosted on an AWS EC2 node predicts the current drawing's icon category.

The web browser sends all the information rendered on canvas to the Amazon EC2 instance after a user completes drawing a UI element by pressing the ”icon done” button. PSDoodle's similarity metric uses element category, shape, position, and occurrence frequency to seek up the top-N matching screens in its dictionary of Rico screen hierarchies [14]. After performing the similarity calculation, PSDoodle sends the retrieved screen names to the website. The website fetches the necessary screens from an AWS S3 bucket and displays them to the user.

4.3.1 PSDoodle's Website

PSDoodle's website is hosted on an AWS EC2 general purpose instance (t2.large) with two virtual CPUs and 8 GB of RAM. PSDoodle provides a guided interactive tutorial (`http://pixeltoapp.com/toolIns/`) for first-time users and a cheat sheet of supported UI elements in the top-left of the main page. To make these instructions self-explanatory, we recruited two UI designers via the freelancing website Upwork[3], recorded their tool usage and incorporated their feedback. This feedback has yielded, e.g., uniform text, position, and size of PSDoodle's buttons (Figure 4.3) plus a refactored cheat sheet (e.g., on when and how to draw a text button, Figure 4.4).

[3] `https://www.upwork.com/`, accessed January 2022.

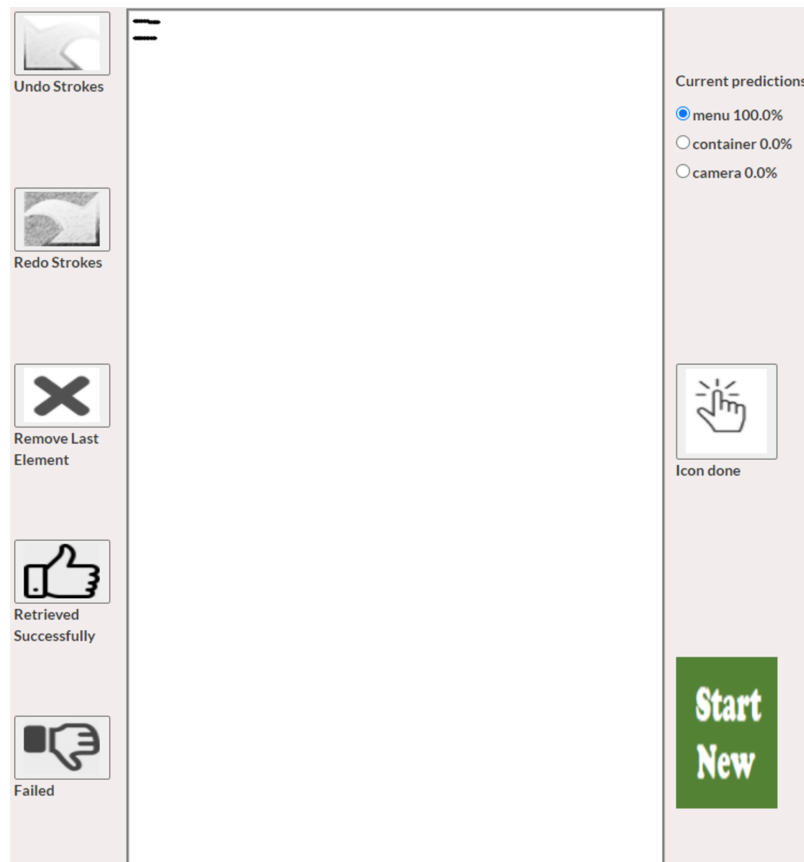


Figure 4.3. PSDoodle drawing UI, under which PSDoodle shows its current top-N Android search result screens (omitted)..

PSDoodle's drawing interface is a canvas element with support for both mouse and touch. PSDoodle provides basic drawing features for interactions. Users can undo or redo strokes and remove the last icon (Figure 4.3 top left). PSDoodle's client-side JavaScript handles all the basic events like touch-start, touch-end, and a stack of strokes to handle redo/undo features. Each time the user adds a stroke to the current icon doodle, the PS-Doodle website detects the touch-end event and sends an HTTP post request with the stroke coordinates to the AWS EC2. PSDoodle's web application parses the response from AWS EC2 and shows its current top-3 three icon predictions (top right). A user can pick any of

these three (and tap “Icon done” or ’d’ on the keyboard) or continue editing the current icon doodle.

After the user adds (or removes) an icon, the website submits a search query containing all recognized UI elements currently on the canvas plus their on-canvas locations. Based on PSDoodle’s similarity metric [14] the website retrieves the top-80 screens’ ids from AWS EC2. The website then retrieves the screens corresponding to these ids from the AWS S3 bucket for display on the bottom of the canvas. When a user clicks on a search result, it shows an enlarged version of the screen on the website. A user can navigate between the next 80 search results via the “next”/“previous” buttons. The user can give feedback to PSDoodle via the thumbs up/down buttons (lower left) and start a new screen search (lower right).

PSDoodle stores two resolutions of each Rico screen in an AWS S3 bucket. PSDoodle uses the low-resolution images to quickly show the search results (in a gallery view). When a user clicks on an image in the search gallery, PSDoodle fetches the higher-resolution image from the S3 Bucket and displays it of the left side on the website.

4.3.2 Recognizing Individual UI Elements

Figure 4.4 gives an overview of PSDoodle’s visual query language, i.e., the 23 graphical primitives PSDoodle recognizes and how the user can combine them to create compound and nested UI elements. According to the number of element labels inferred by Liu et al. [42], DoodleUINet [12] covers several of the most popular UI elements in Rico. PSDoodle uses 7 QuickDraw classes that were a suitable match for UI sketching. PSDoodle provides placeholders for text (squiggle line), image (jailwindow), and for not directly supported icons (cloud). Placeholders help the user to avoid fine details of text and images.

PSDoodle provides options to draw compound UI elements. For example, the Android text button is one "text" inside a "square" drawn separately. Sketching one UI element at a time permits users to nest UI elements within a container by drawing them separately.

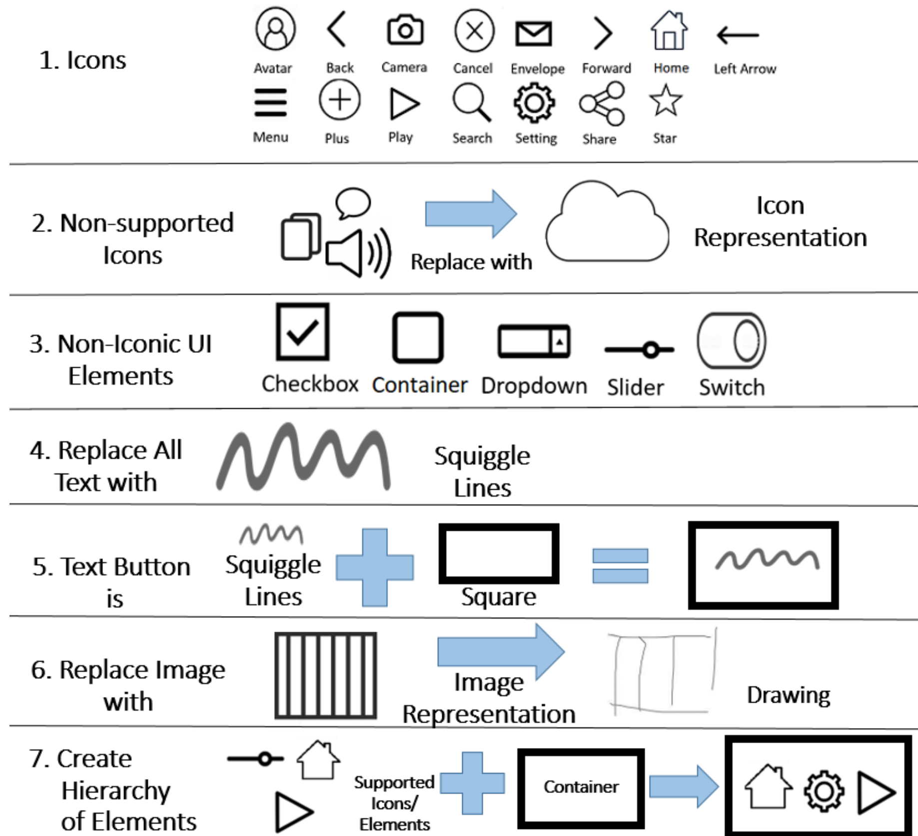


Figure 4.4. Cheat sheet PSDoodle shows to users: 23 graphical primitives plus compound and nested UI elements..

PSDoodle uses a deep neural network similar to QuickDraw's network architecture [44] to identify UI element class from strokes. Three 1-D convolutional neural networks (CNN) layers followed by three Bi-LSTM layers, and a fully-connected layer make up PSDoodle's deep neural network. PSDoodle used DoodleUINet (600 doodles for each of its 16 classes) and a random 600-doodle sample of each of PSDoodle's 7 QuickDraw classes to train the deep neural network and yielded a 94.5% test accuracy.

We adopted a faster TensorFlow implementation of the deep neural network in AWS EC2 that avoids regenerating the network graph for predicting class labels for each stroke. PSDoodle detects the class label, confidence score with the Tensorflow implementation, and shows them instantly (in under a second). PSDoodle guides a user to express their drawing intention by instantly showing the updated prediction with each stroke.

4.4 Exploring PSDoodle's Usage

Following the most closely related work [34, 35], we evaluated screen search performance by measuring top-k (screen) retrieval accuracy [14]. We thus showed a participant a target screen to sketch and measured where in the result ranking the target screen appears. Top-k retrieval accuracy is the most common metric for sketch-based image retrieval tasks and correlates with user satisfaction [46].

While PSDoodle's top-10 screen retrieval accuracy of 88% is similar to the state-of-the-art's 90%, PSDoodle cuts the state-of-the-art's screen retrieval time in half [14]. In this section, we explore how PSDoodle supports different sketching styles, how many of the tool's top-10 search results are relevant to the user's query sketch, and how users have described PSDoodle.

4.4.1 Supporting Different Sketching Styles

Table 4.1 shows UI category drawings with different numbers of strokes in the test dataset, showing a variety of QuickDraw's game participants' and DoodleUINet crowd-workers' drawing styles. PSDoodle can thus detect sketches with high confidence for different drawing styles.

As an example, in all cases when PSDoodle classified a slider test sketch that consists of five strokes, it assigned a 100% confidence that the sketch is indeed a slider. But PS-Doodle similarly had correctly classified (with high confidence) other slider test sketches,

e.g., those consisting of four or three strokes. The likely reason for this behavior is that PSDoodle's classifier has been trained on sketches from a variety of people. Similarly, adding more training samples from a wider variety of crowdworkers may make PSDoodle even more robust to different drawing styles.

Table 4.1. Average confidence PSDoodle has in a sketch of the given total stroke count belonging to sketch's target category. For example, the network has 50% confidence on average for completed avatar sketches of a total of 7 strokes to be avatars.

Cat.	**Confidence by sketch's total strokes**								
	1	2	3	4	5	6	7	8	9+
Camera	-	92	96	96	100	100	-	100	-
Cloud	-	-	75	97	95	100	89	89	100
Envel.	100	96	100	100	86	100	-	-	100
House	80	97	94	100	72	100	100	100	50
Jail-win	89	78	100	100	67	0	-	-	-
Square	98	100	100	75	-	-	-	-	-
Star	99	100	100	100	100	-	100	-	-
Avatar	-	100	82	96	89	100	50	100	-
Back	97	100	0	-	-	100	-	-	-
Cancel	-	100	77	50	50	-	-	-	-
Checkb.	100	96	61	71	83	0	100	100	67
Drop-d.	-	100	98	88	100	100	100	100	100
Forward	100	100	-	-	-	-	-	-	-
Left-arr.	80	87	92	-	100	-	50	-	-
Menu	-	100	100	100	100	100	-	-	-
Play	96	96	95	100	100	-	-	-	0
Plus	-	100	93	100	100	-	100	0	-
Search	100	98	100	100	100	-	-	-	100
Setting	100	94	92	67	100	50	-	100	83
Share	-	0	-	-	100	100	98	100	100
Slider	100	94	97	100	100	100	100	-	-
Squiggle	98	83	100	100	100	100	-	-	0
Switch	100	97	80	91	100	100	100	50	0

4.4.2 Surfacing Several Relevant Result Screens

Figure 4.1 shows an example partial screen sketch and PSDoodle's top 5 search results. The search results are of high quality as the result screens contain all sketched UI elements at about the location they appear in the query sketch.

In a user study with 10 participants[4] we received similar feedback. For each of a total of 34 screen sketches, participants judged the quality of each of PSDoodle's top-10 result screens. Participants judged 145/340 of these screens as relevant to their search query.

4.4.3 User Experience

Nine users have filled out a brief survey about their experience, including the following two open-ended questions.

Q1 What improvement/features do you suggest to make the interface better for a user?

Q2 How was the overall search results?

While all answers are available[5], following are a few highlights. Regarding improvements (Q1), users asked for more icon support. Supporting more UI element categories is mostly a matter of gathering additional training samples and retraining the deep neural network. Regarding the overall search results (Q2), users were generally positive. Following is one quote:

"good, the results were similar to what I was looking for".

Another user said the following.

"I tested different shapes the overall result was good. [..] I was overall satisfied with what I tested."

[4]We recruited 10 Computer Science students who had no prior UI/UX design experience. Each participant first spent on average some 9 minutes in PSDoodle's interactive tutorial (`http://pixeltoapp.com/toolIns/`). We then gave Rico target screens to sketch.

[5]`https://github.com/soumikmohianuta/PSDoodle/blob/master/ComparisonResult/ToolSurvey/PSDoodle_Tool_Survey.pdf`

4.5 Conclusions

Current approaches to searching through existing repositories are either slow or fail to address the need of novice users. Interactive partial sketching, which is more structured than a keyword search and faster than complete-screen inquiries, is a viable option. PS-Doodle is the first tool to offer interactive screen search with sketching and live search results.

CHAPTER 5

Searching Mobile App Screens via Text + Doodle

5.1 Introduction

Traditionally, When people want to find mobile app examples, they usually use keywords to search for it on tools like Google's image search[55, 56]. However, it can be tough to describe the structure of a screen accurately with just words. Sketch-based search presents a structured approach for representing element structure with greater ease. Through this approach, users can search mobile screens by creating visual representations of either the entire screen or a portion [14, 34, 35].

A complete screen drawing-based solution can be tedious, slow, and lack interactivity. Partial sketching as an interactive alternative for search has drawbacks in terms of precision and can yield inconsistent outcomes depending on the user's proficiency. [14]. The sketch-based search method has yet to be widely embraced and still encounters obstacles in identifying textual elements on the screen or differentiating between two text buttons with varying content. No attempt was made to merge these two distinct search approaches.

As mobile applications gain widespread use, companies allocate significant resources towards developing user interfaces that align with user experience standards [4, 5, 57]. A search engine that efficiently and accurately locates mobile app screens can offer substantial benefits for user interface design, such as facilitating requirement gathering, trend analysis, feature analysis, developer creativity, and evaluation benchmarks. Consequently, an effective mobile app screen search engine can significantly impact software developers and end-users.

TpD represents a novel approach that combines sketch and text to search for mobile screens. TpD features a digital drawing interface similar to PSDoodle, which allows users to draw using a touch or a mouse. Moreover, TpD provides keyword-based searching for mobile screens. By integrating word and doodle queries, TpD retrieves more relevant mobile app screens from a pool of 58k Rico screens [16].

The image in Figure 5.1 depicts an example query session of TpD. The user initially draws a 'menu' icon on the top left corner of the canvas in the TpD website. After completing the drawing of the 'menu' icon, TpD retrieves the top 50 best matching mobile screens by iterating through 58k Rico screens and displays them at the bottom of the canvas within 2 seconds. The first row of the image in Figure 5.1 highlights the top 5 screens of TpD, and all of them have a 'menu' icon situated near the area where the user sketched it. Next, the user adds a 'search' icon doodle in the top right corner of the canvas, and TpD promptly updates the search result within 2 seconds. The second row of the image in Figure 5.1 displays the updated top 5 screens, and all contain both the 'menu' and 'search' icons in their respective positions. Afterward, the user includes a text query using the term 'Editor' and marks its position as top-left on the screen with the syntax 'tl: Editor.' The third row of the image in Figure 5.1 showcases the top 5 results retrieved by TpD, all of which feature the text 'Editor' in the top-left corner, along with the 'menu' and 'search' icons situated in the positions specified by the sketch query. Finally, the user adds another text query, 'necklace,' prompting TpD to update the search results. The last row of the image in Figure 5.1 displays the top 5 search results.

PSDoodle's query can be keywords, sketch, or both. TpD incorporates PSDoodle's deep learning techniques to recognize sketched UI elements on the drawing interface. TpD also adopts PSDoodle's algorithm to compute a ranking score for a Rico screen based on UI element type, position, and shape. For keyword query, TpD matches terms with the mobile screen's visible text and UI element descriptions while accounting for synonymous

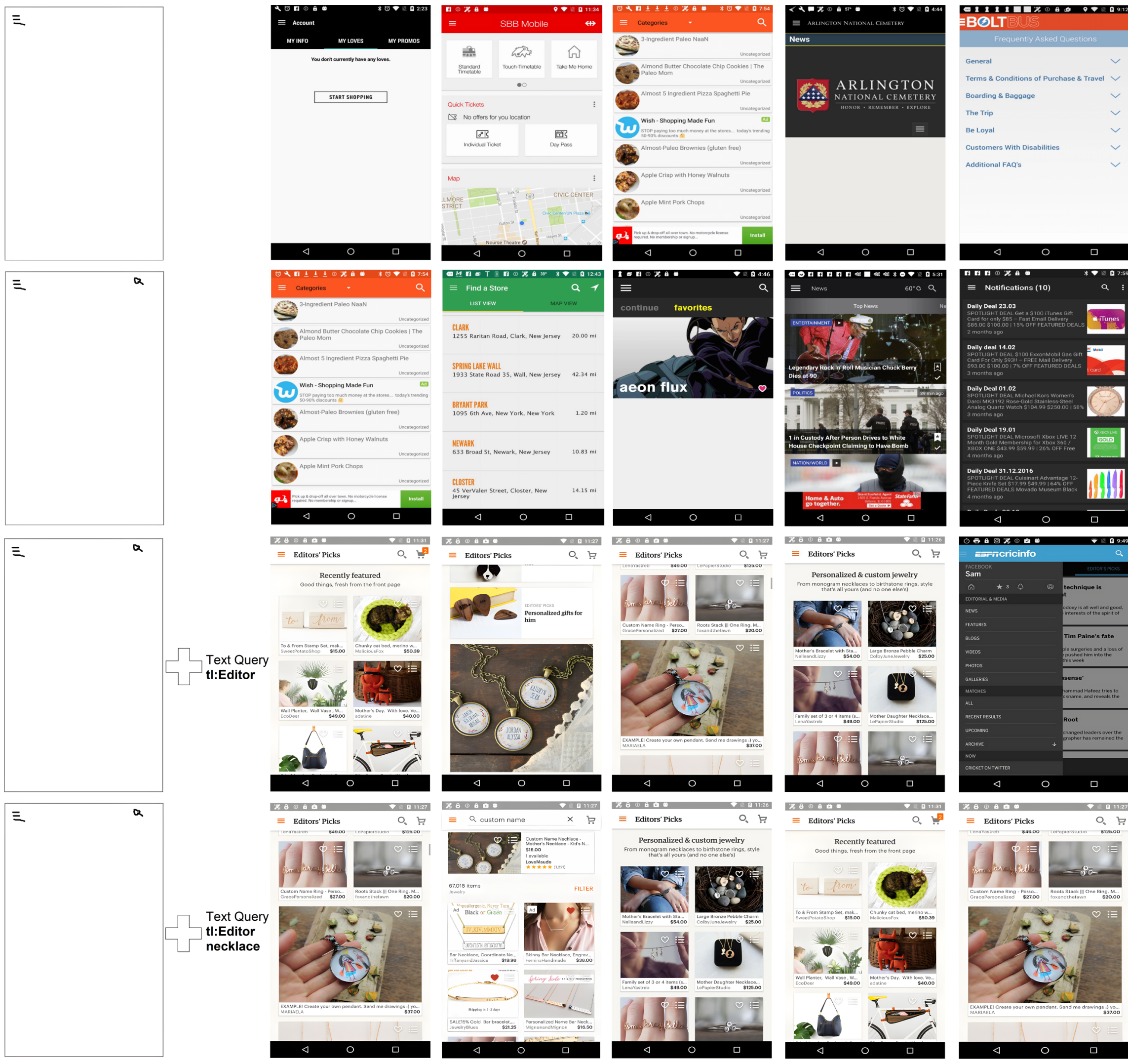


Figure 5.1. A screen query session with TpD and the first column of each row displays the search. The subsequent columns in each row show the top-5 search results (in order) out of 58k Rico screens returned by TpD. All screens exhibit the sketched UI elements in roughly the exact location as the user drew them. When the user adds text to the query, the updated search results show screens containing the sketched UI elements and the specified text in their respective locations. Within each row, TpD returns the top 50 ranked result screens in just 2 seconds, including the round-trip duration from the user's device to the AWS-hosted TpD. .

and semantically related words. The tool is invariant to inflectional or derivational-related changes in the text. TpD aware of the position of the text on the mobile screen, allowing

users to annotate a text query with the 'position: keyword' syntax. The position can take one of the 12 values (l: left, r: right, t: top, b: bottom, lt/tl: top-left, rt/tr: top-right, bl/lb: bottom-left, br/rb: bottom-right). TpD fetches search results in real-time for any update in query, providing an interactive and iterative approach to screen search.

We asked 10 participants who were new to the TpD to search for a specific Rico mobile screen using text and sketches. TpD successfully retrieved and displayed the target screen in its top-10 search results 90% of the time, with the target screen consistently appearing on the first page (each page exhibits 50 mobile screens) at the end of the query. The average query length was 4, and the average search time was 45 seconds. These outcomes showcase significant progress in performance compared to the closest related tool, PSDoodle [14, 15]. Moreover, the achieved accuracy levels are on par with those reported in work by Swire[34]. This paper offers several significant contributions-

- TpD is the first tool, that interactive and iterative search system for mobile screens that incorporates both text and sketch input. Users can access the tool at the following URL: `http://pixeltoapp.com/WnD`.
- In comparison to the state-of-the-art, TpD exhibits equivalent top-10 search accuracy while significantly outperforming in terms of speed.
- The source code, processing scripts, training data, and experimental results of TpD are accessible under open-source licenses. [1]

5.2 Background

5.2.1 Rico: Corpus of 72k Android App Screens

Rico, a dataset compiled by Deka et al. [16], was obtained by mining 72k Android app screens using automatic exploration techniques and crowd-sourcing. Using runtime

[1] https://github.com/DoodleUI/WnD

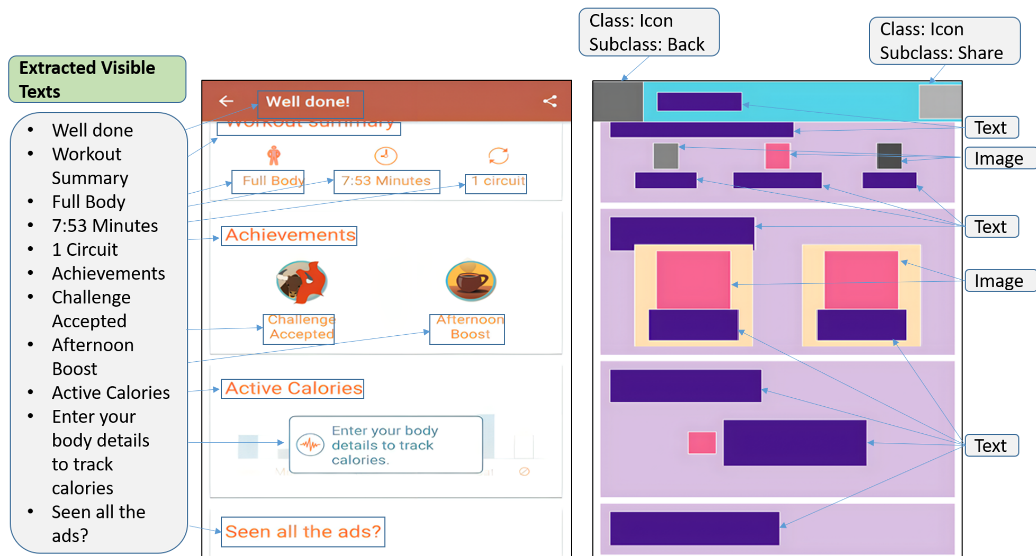


Figure 5.2. The Rico dataset contains all visible text within mobile screens' user interface (UI) hierarchy (depicted on the left). Liu clustered UI elements of Rico mobile screens into 27 categories and sub-categories based on their properties and assigned a descriptive term to represent them (illustrated on the right). .

information of each screen, the researchers extracted the hierarchy of UI elements and attributes of each component, such as its position, Android class, displayed text, etc. Figure 5.2 shows an example Rico mobile screen.

Liu et al. [42] clustered the UI elements of Rico screens based on factors such as image similarity, surrounding text snippets, and code-based patterns. They divided UI elements into 25 categories: "Text", "Icon", "Image", etc. Additionally, they sub-categorized icons into 135 classes and text buttons into 197 types. The study further assigned a description representing the UI element category and its sub-category. Three mobile application developers reach a consensus to select the most appropriate word description representing a specific UI element category to ensure the chosen description's accuracy. Figure 5.2 illustrates examples of word descriptions for the element class and icon sub-category class of a mobile screen.

5.2.2 Swire: Search by Full-screen Sketch

Swire, one of our competing systems, utilizes a complete mobile screen sketch to identify similar Rico mobile screens. Swire collected a 3.8k low-fidelity full-screen Rico screen sketches dataset from four experienced UI designers. They then trained a deep neural network on a subset of this dataset consisting of 1.7k Rico sketch-screenshot pairs. The trained network achieved a top-10 screen retrieval accuracy of 61%. Swire's approach emulates a traditional paper-based design style, where users sketch with pen on paper inside an Aruco marker frame.

Each sketch must go through several pipelines, like scanning and adjusting projection. However, changing a sketch using this approach requires starting over, scanning, and following processing steps can be time-consuming. A recent study on Swire has achieved a top-10 accuracy of 90.1%, currently considered state-of-the-art [35]. Swire sketches follow a predefined drawing convention, where users replace any text with a template "Lorem ipsum dolor" or squiggly lines. Additionally, the deep neural networks employed by Swire are agnostic to the actual content of the text.

5.2.3 PSDoodle: Search by Icon Doodles

Our work is closely related to PSDoodle [14, 15], an iterative and interactive tool for searching mobile screens from incomplete, partial drawings. PSDoodle supports drawing 24 of the most common Android UI elements. PSDoodle divides the mobile app screen into 24 equal-sized tiles (6 along the width and 4 along the height) to match UI elements in different scales. PSDoodle uses the UI element hierarchy of 58k Rico mobile screens to construct a dictionary that maps each element type to a list of mobile screens. Each screen in the dictionary lists the percentage of the tile area covered by how many instances of that UI element type. With the help of this dictionary, PSDoodle can search through 58k Rico screens and find the best match in real-time.

PSDoodle's deep neural network trained on DoodleUINet [12] and QuickDraw [58] recognizes UI element category from drawing strokes. When the user indicates correct recognition of a UI element, PSDoodle's algorithm retrieves corresponding Rico screens information from its dictionary. Then assigns a score based on the UI element's type, position, and shape. PSDoodle sorts the mobile app screens by the score to find the top match.

The PSDoodle website is hosted on AWS and retrieves the top-match mobile screens with a delay of 2 seconds. With a user evaluation, PSDoodle reported top-10 accuracy of 88.2% and an average query session of 105s. PSDoodle visual language allows placing a common placeholder for text (squiggle line), not considering any text that appears on the screen. The top-10 accuracy of PSDoodle is on par with the current state-of-the-art. Nonetheless, in two out of 34 evaluation sessions, the PSDoodle failed to display the target screen on the website.

5.3 Overview and Design

Figure 5.3 provides an overview of TpD's architecture. TpD adapts PSDoodle's architecture (Figure 5.3's grayed elements) to rank a corpus of mobile app screens by their similarity to a query of sketched icons. TpD adds to that by collecting from the Rico dataset [16] mobile screens' visible text and via Liu's work [42] textual descriptions of screens' UI elements. The extracted texts undergo multiple preprocessing steps to improve retrieval performance (Figure 5.6). Finally, TpD uses Elasticsearch [59] to index the texts for faster retrieval. TpD calculates screen scores for text and sketch queries separately, followed by normalization and merging of these scores to determine the top-N matching screens.

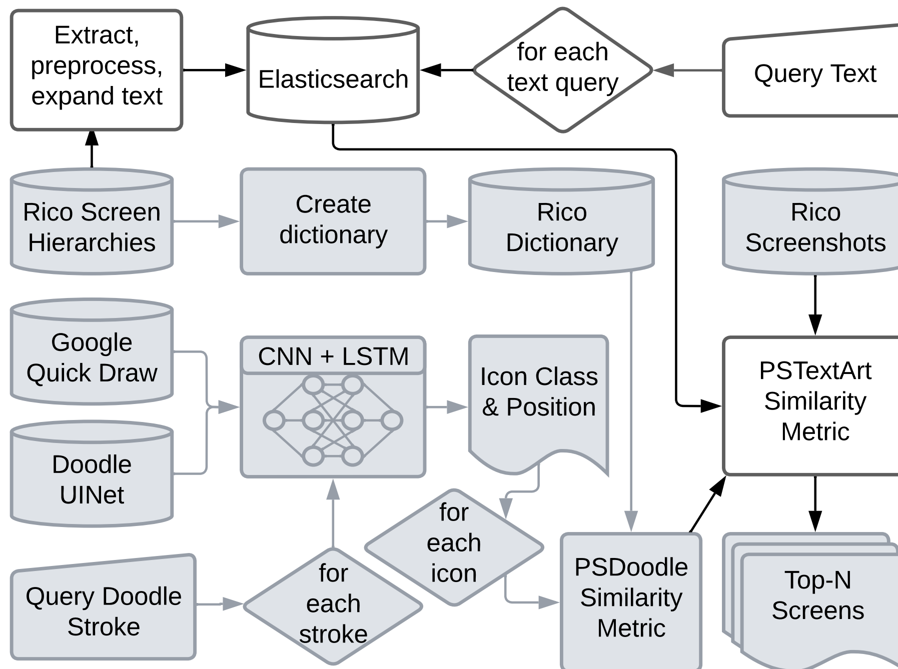


Figure 5.3. TpD architecture (grayed parts adapted form PSDoodle): To search, a user submits a series of text and icon doodle queries on the TpD site (`http://pixeltoapp.com/WnD/`), which communicates with its back-end on AWS. .

5.3.1 Screen Search Via Icon Doodles & Positional Text Queries

Figure 5.4 shows the website of TpD. TpD displays the current top-3 predictions for UI elements (top right) when a user draws on the canvas. Users can add or remove text in the query set through the search bar above the canvas in TpD. In the Figure 5.4 example, the user has already issued two text queries, “facebook” and “tl:twitter”.

Figure 5.5 is the query language of TpD represented to the user as a cheatsheet on its website. Users can draw any atomic UI element, or combine them to create a complex UI element and establish a hierarchy, per the instructions provided in Figure 5.5. Upon finishing the current UI element drawing on the canvas, a user signals the tool by clicking the “Icon done” button.

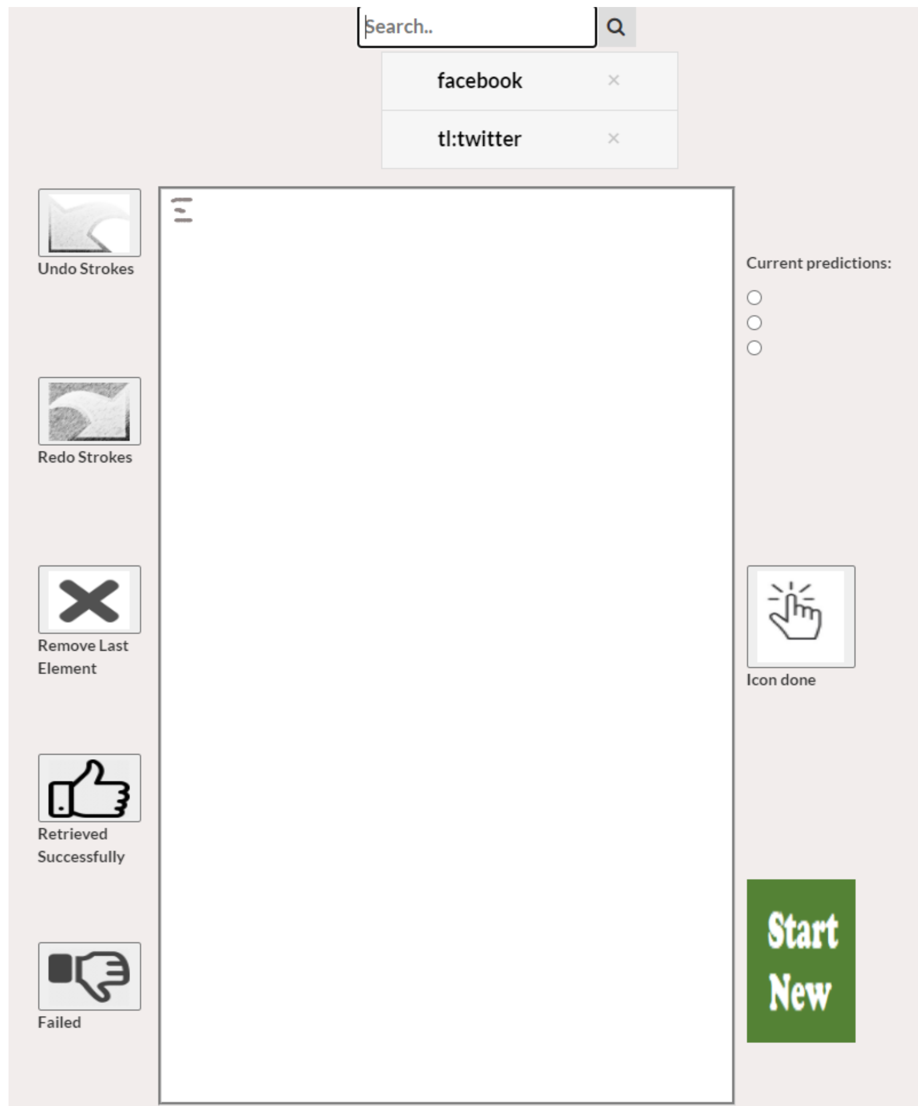


Figure 5.4. Query interface of TpD web site: Text input area (top) with two submitted queries ("facebook" and "tl:twitter") and sketch input area (bottom) with one submitted query (menu icon). The website also displays TpD's current top-N Android search result screens (cut from screenshot)..

To allow finer-grained text queries, TpD allows the user to specify where a given query text should appear on a screen. This enables users to label text with its position on the screen, facilitating a more precise and targeted search and yielding more accurate results. TpD currently allows the user to specify one of four screen quadrants, top-left

(via prefix “tl:” or “lt:”), top-right (“tr:”/“rt:”), bottom-left (“bl:”/“lb:”), and bottom-right (“br:”/“rb:”). Alternatively, TpD allows the user to specify the combination of two adjacent quadrants, i.e., top (“t:”), bottom (“b:”), right (“r:”), or left (“l:”), yielding a total of 12 positional keywords for text queries.

5.3.2 Screen Texts, UI Descriptions, And X/Y Pixel Coordinates

We extract two text types from each of the 58k Rico mobile app screens. First, we parse the screen’s hierarchical UI container tree structure to extract all text displayed to the user. On the left of the Figure 5.2 example, this would include the strings listed in the left side-bar, such as “Well done!” and “Full Body”.

Second, from the same hierarchical UI container tree we extract all UI elements and their descriptions according to the similarity-based clustering by Liu et al. [42]. On the right of Figure 5.2, examples of this type of text describing the screen are in the call-out boxes such as “Back” and “Share”.

Together with both types of text, we also extract from the hierarchical UI container tree the x/y pixel coordinate location of where the text or UI element appears on the screen. While earlier work has already inferred these texts and on-screen locations for the 58k Rico screens, applying the same techniques to further third-party apps would be straightforward.

5.3.3 Screen Contents’ Stop Words, Names, And Lemmatization

To support a wide range of possible user text queries, TpD uses two separate text preprocessing pipelines, one for screen text and the other for UI element descriptions. The involved preprocessing steps are used to improve performance in many natural language processing (NLP) use cases (e.g., in sentiment analysis, document categorization, and document retrieval for user queries [60, 61]).

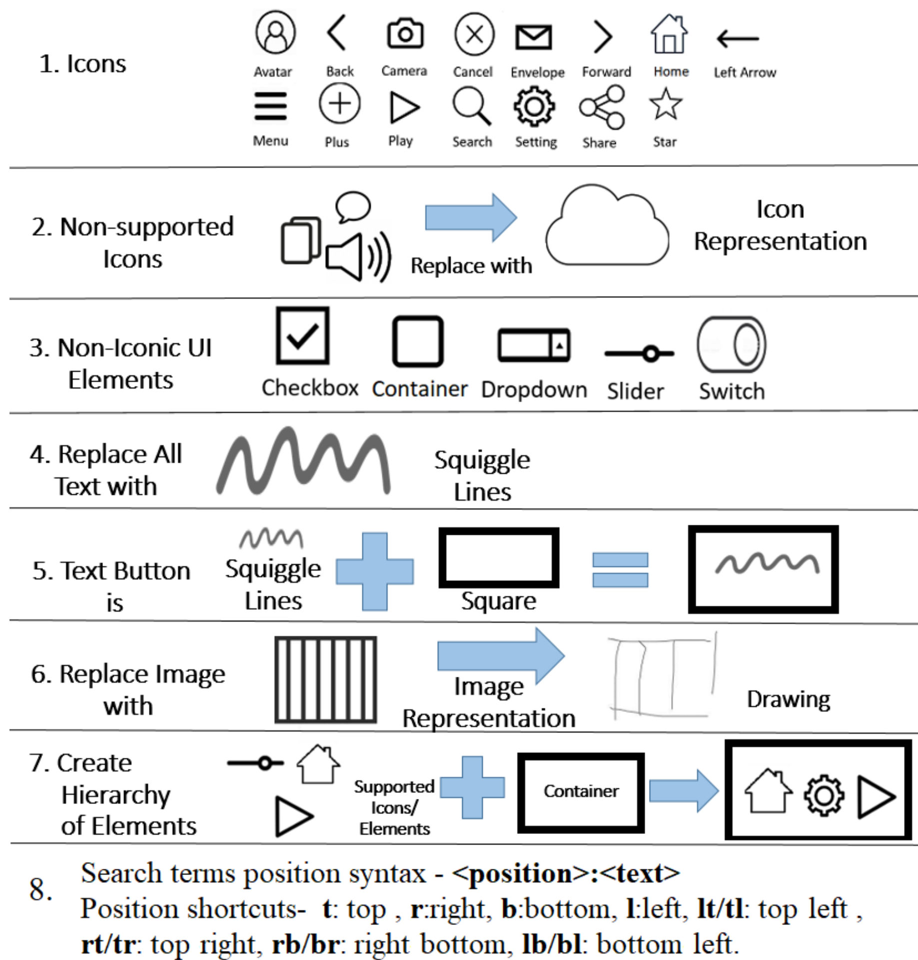


Figure 5.5. TpD describes its query language to users via this on-screen “cheat sheet”. .

The main difference between TpD’s two preprocessing pipelines is the treatment of stop-words (which in general carry little distinctive information [33]). While TpD identifies and removes stop-words from on-screen text using the Natural Language Toolkit [62], TpD preserves stop-words in the UI element descriptions. There stop-words such as “up” and “down” have distinct meaning that differentiates one UI element from another (e.g., arrow up vs. arrow down).

A simple way to support a wide variety of user queries is to rewrite both the corpus and query via stemming [63, 64], which using rules maps each word to its root form (e.g.,

"walked" or "walking" to "walk"). To also consider the word's context and part of speech within a sentence, TpD instead maps a word to its morphological root via lemmatization [65]. This yields a more precise root form, such as "bet", or the retention of the original form, such as "better", depending on the word's part of speech and context.

TpD uses spaCy's en_core_web_trf-3.4.1 English language model [66], which has excellent accuracy in identifying a word's part of speech in a sentence (precision 0.98), segmenting sentences (precision 0.96), and recognizing named entities (precision 0.9) [67]. TpD stores a word's lemma form in lowercase and consults spaCy to determine if a word is a named entity (such as a person, organization, or location). TpD's repository includes the resulting preprocessed texts for the 58k Rico screens.

5.3.4 Adding Screen Content Synonyms

While lemmatization covers a variety of user queries, it does not cover synonyms. We thus add the synonyms of our screen contents by combining automatic query expansion (AQE) approaches. For background, AQE techniques include lexical and contextual analysis, relevance feedback, and supervised approaches [68]. As relevance feedback and supervised methods rely on user feedback and search history, and we are unaware of public corpora on mobile screen content queries, TpD focuses on lexical and contextual analysis. Lexical approaches encode basic thesaurus-like word relations (e.g., synonym, hypernym, and hyponym).

Contextual analysis infers such word relations from features such as two words tending to appear together in a large variety of documents. The contextual analysis employs unsupervised deep learning on a large corpus of unstructured text and typically encodes each word's relations to other words as a fixed-length vector of real numbers (aka word embedding) [69, 70], positioning contextually related words close to each other in vector

space [69, 71]. Then word similarity can be measured via vector distance metrics (e.g., l1, l2, and cosine).

To adapt existing contextual models to mobile app screen contents, we tried fine-tuning the pre-trained Word2vec [69] and GloVe[71] word embedding models with the local context of our preprocessed Rico mobile app screen contents. However, we quickly realized this not to work well, as each Rico screen on average only contains four words and the screen contents are often disjoint. (e.g., while in close proximity of each other on the Figure 5.2 screen, the two text snippets "enter your body details to track calories" and "seen all the ads?" represent two different app features). For instance, this yielded a fine-tuned Word2vec [69] model listing the two most common related words of "avatar" as "edit" and "too".

Word2vec, GloVe, and FastText [70] represent a variety of contextual approaches and training sets. Word2vec (word2vec-google-news-300 of 3M words and phrases trained on 100B words from Google News) focuses on language-specific information, GloVe (glove-wiki-gigaword-300 of 400k words trained on 5.6B tokens from Wikipedia 2014 + Gigaword) uses global statistical data, and FastText (fasttext-wiki-news-subwords-300 of 1M words trained on 9B+ words) is reliable for out-of-corpus words [72]. To minimize bias and improve word coverage, TpD combines all three models as follows.

First, for each screen content word, we retrieve the 10 most-similar words from each of the three models via cosine similarity (as cosine considers vector orientation and provides better matches than other metrics such as Euclidean distance [73]). We thus retrieve three sets of up to 10 synonyms each, each synonym with its similarity [0..1]. We then merge the three sets, adding overlapping words' similarity scores and thus favoring common synonyms.

In the resulting ranked list, we break ties via our two lexical models WordNet [74] and ConceptNet [75], demoting a word that is not in WordNet, and to break a further tie de-

moting a word that is not in ConceptNet. Following earlier work [76], we use WordNet [74] as it covers English well and works well for query expansion [77, 78] and ConceptNet for its knowledge graph of 5.6M nodes and 34M+ edges.

If a word is not in any of the three contextual models, we again fall back on our lexical models and first look up the word's synonyms in WordNet and then in ConceptNet. This procedure has yielded three synonyms for each word in or describing the 58k Rico screens. The exceptions are the 7,448 unnamed entities, as users are typically interested in the entity itself and not a similar entity. The resulting "Synonym.txt" file of 29,503 screen content words and their 88,509 synonyms is in TpD's repository.

5.3.5 Storing Screen Contents' Locations And Synonyms

To support positional text queries, TpD divides each screen into a 2x2 grid of equal-sized rectangles and then maps each extracted screen text or UI element description to one of these four quadrants. While the current TpD implementation maps a screen content's top-left corner's x/y coordinate, this could easily be generalized to mapping the screen content to each quadrant it overlaps with.

TpD stores and indexes its preprocessed screen contents and their on-screen locations together in Elasticsearch. Elasticsearch is widely used for data storage and search [59, 79, 80]. Specifically, for each word on each Rico screen, TpD maps a pair (words and the phrase's screen quadrant (aka document zone)) to the screen's unique ID. For each of the (pre-processed) words, TpD further updates Elasticsearch's synonym mapping from that word's pre-processed form to the three synonyms we inferred earlier.

We further configure Elasticsearch to match each query word with Levenshtein edit distance one (i.e., allowing one one-character change per query word). For instance, screen content word "setting" would match when inserting (e.g., "settiing"), deleting (e.g., "seting"), or replacing (e.g., "setling") one character. Choosing edit distance one is a trade-off

between allowing more query typos vs. minimizing query response time. TpD runs its website and Elasticsearch on the same AWS EC2 general-purpose computing instance (t2.large: two virtual CPUs and 8 GB of RAM).

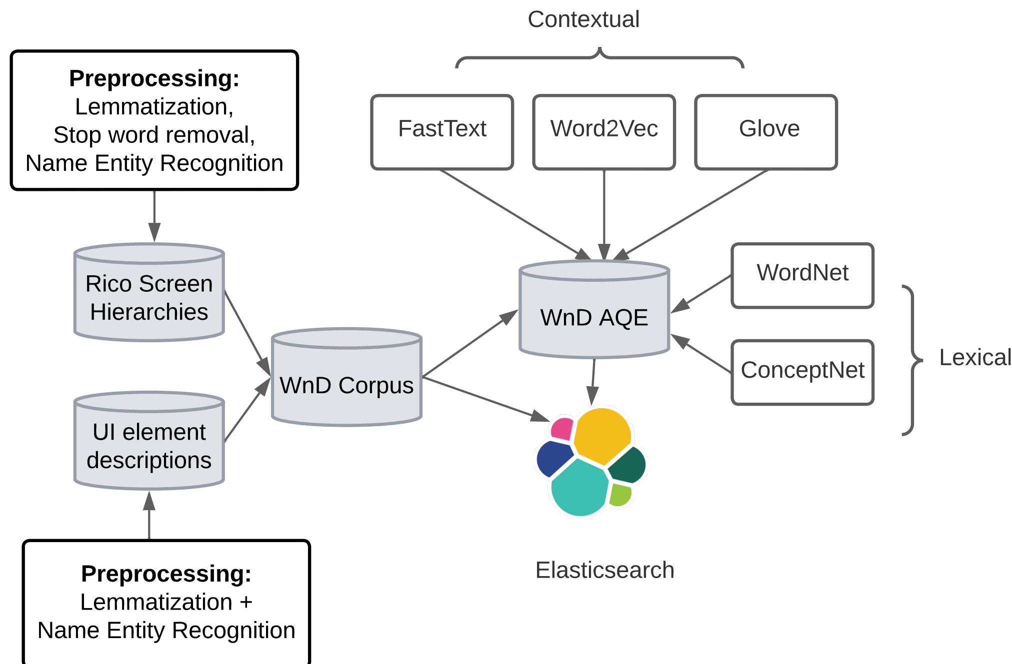


Figure 5.6. TpD extracts visible texts from Rico's screen hierarchy and UI element description from Liu's work. Each text then goes through preprocessing pipelines to create the TpD corpus. TpD uses lexical and contextual analysis to find each word's top 3 extended terms and store them in the Elasticsearch database. Elasticsearch stores all text indexed by Rico's mobile screen id and position for faster retrieval..

5.3.6 Drawing Recognition With TpD

To recognize as a UI element the stroke sequence the user draws with a mouse or touchscreen, TpD uses PSDoodle's deep neural network architecture [14], i.e., a 1-D convolutional neural network (CNN) layer with 48 filters and kernel size 5, followed by a 1-D CNN layer with kernel size 5 and 64 filters, a 1-D CNN layer with kernel size 3 and 96 fil-

Table 5.1. Doodle recognition trained on 80% classifying the other 20% (the test doodles): 1st stroke at which a tool ranks a doodle's correct class first (top-1); all/fin = test doodles' correct class not ranked first in each/final stroke; cnt = count; avg = average; m = median; l = min; h = max; std = standard deviation.

Category	20%	PSDoodle (non-uniform sampling)							TpD (after re-sampling)						
	cnt	1st stroke to top-1					cnt		1st stroke to top-1					cnt	
		avg	m	l	h	std	all	fin	avg	m	l	h	std	all	fin
Camera	143	2.40	2	1	5	0.77	7	10	2.61	2	1	5	0.85	1	6
Cloud	154	1.06	1	1	2	0.23	2	9	1.07	1	1	2	0.26	3	5
Envelope	145	1.87	2	1	6	0.87	0	7	1.89	2	1	6	0.89	1	4
House	135	2.13	2	1	5	0.98	0	1	2.25	2	1	9	1.22	3	7
Jail-window	143	3.34	3	1	8	1.11	0	7	2.58	2	1	7	1.02	0	5
Square	147	1.12	1	1	3	0.34	0	2	1.12	1	1	4	0.42	1	9
Star	148	1.15	1	1	6	0.57	0	2	1.12	1	1	3	0.39	1	3
Avatar	136	2.87	3	1	7	0.92	2	3	2.79	3	1	5	0.71	1	3
Back	122	1.06	1	1	3	0.27	2	3	1.02	1	1	3	0.18	2	1
Cancel	127	1.96	2	1	3	0.57	1	1	1.93	2	1	3	0.44	0	1
Checkbox	134	2.24	2	1	6	1.04	2	1	2.30	2	1	6	1.15	0	2
Drop-down	133	2.49	2	1	9	1.19	7	5	3.40	3	1	8	1.21	3	7
Forward	122	1.05	1	1	2	0.22	0	3	1.04	1	1	2	0.20	0	3
Left arrow	123	1.96	2	1	3	0.56	0	7	1.89	2	1	3	0.48	0	1
Menu	126	2.16	2	2	8	0.63	1	3	2.91	3	2	6	0.44	0	1
Play	127	2.03	2	1	7	0.93	3	21	2.03	2	1	5	0.84	2	3
Plus	120	1.43	1	1	3	0.59	2	7	1.39	1	1	4	0.67	0	6
Search	122	1.89	2	1	5	0.51	3	2	2.02	2	1	5	0.37	0	1
Setting	111	2.66	2	1	10	1.71	0	3	2.93	2	1	27	3.11	1	8
Share	117	2.97	3	1	7	1.15	3	2	3.41	3	2	8	0.88	1	1
Slider	134	1.85	2	1	6	0.70	0	4	1.86	2	1	4	0.62	0	2
Squiggle	144	1.08	1	1	3	0.29	0	0	1.12	1	1	4	0.53	0	0
Switch	137	2.59	2	1	7	1.20	0	8	2.59	2	1	8	1.09	1	5

ters, 3 Bi-LSTM layers, and a fully-connected layer. Each stroke is a sequence of lines and a line is given by its endpoint x/y coordinates.

TpD trained on sketches from 7 icon classes from QuickDraw and 16 from DoodleUINet. Unfortunately, QuickDraw and DoodleUINet collected sketches using different drawing interfaces and thus different point sampling ratios. In other words, QuickDraw and DoodleUINet would represent the same stroke via different points (and thus lines). These

different sampling ratios make it harder for a deep neural net to learn the icon classes' underlying structure, yielding non-optimal icon sketch recognition performance.

Prior work has resampled sketches to make line endpoints equidistant [81]. In contrast, for each stroke TpD determines the Euclidean distance between all pairs of line endpoints and uses 20 times the biggest such distance as the stroke's new number of line endpoints. TpD then resamples each stroke to this new line endpoint count using NumPy's one-dimensional linear interpolation. While PSDoodle reported a 94.2% sketch recognition accuracy, after re-training the network on our resampled sketches for 18,131 steps, TpD achieves 95.8% accuracy on PSDoodle's test data. The achieved level of precision mentioned in this study is comparable to that of the Doodle2App [13], albeit across a more extensive range of categories.

Table 5.1 compares icon doodle recognition performance with and without TpD's resampling. Although the average 1st stroke at which TpD ranks a doodle's correct class remained largely unchanged, the resampling pipeline reduced the number of misclassifications at both the final and all strokes. For example, while PSDoodle did not rank camera top-1 for 7/143 camera sketches in each stroke and 10/143 in the final stroke, these ratios reduce to 1/143 and 6/143 for TpD. These findings suggest that the addition of a resampling pipeline has a positive impact on TpD's doodle recognition accuracy.

5.3.7 Ranking Mobile Screens

When the user adds or removes an icon sketch or adds or changes a query text phrase, TpD re-scores each of its 58k Rico screens by how closely the screen matches the new combination of icon sketches and text queries. First (upper half of Algorithm 2), TpD scores each screen based on how closely it matches the size and location of the doodles of one icon class at a time. TpD retrieves these screen scores from PSDoodle and then normalizes these scores to ensure that each sketch element can contribute equally.

In the second phase, TpD starts processing text queries by first tokenizing each query, where tokens are separated by a non-empty sequence consisting of whitespace (" "), dot ("."), comma (","), exclamation mark ("!"'), and question mark ("?") characters. Within the tokens, TpD then checks for one of 12 positional keywords and maps it to Elasticsearch fields. For example, "lt:" or "tl:" specifies a screen's top right quadrant. Each of the four more general keywords "t:", "b:", "l:", and "r:" TpD maps to its two underlying quadrants, e.g., "t:" to top-left and top-right. Finally, if a text query has no positional keyword, TpD maps the query to all four quadrants.

TpD then performs basic stemming (removing prefixes or suffixes), converts each word to lowercase, and sends the resulting query to Elasticsearch. Elasticsearch matches the query in the corresponding position fields and returns the resulting screen IDs and their query score. TpD configured Elasticsearch to calculate scores with weights 10 for exact match and 4 for matching a synonym. TpD then scores all screens one text query at a time and adds these (normalizes) scores to the screens' overall scores. Finally, TpD sorts screens by descending scores, yielding the screens' search result ranking.

To offset the normalization effect, we adjusted PSDoodle's five hyperparameters. To find their optimal values that would result in a high score and top rank for the target screen with the new algorithm, TpD used scikit-learn's GridSearchCV [45] to perform a comprehensive search on the 30 sketches PSDoodle used and published online. The resulting new hyperparameter values are $p_1 = 11$, $p_2 = 1$, $p_3 = 1$, $\Delta_w = 0.7$, and $C_w = 12$.

Algorithm 2 TpD calculates the screen score for the doodle and text queries separately, then adds the normalized scores; sketch = list of maps from icon class to doodles of that class; search(doodles) = score screens by how closely they match the size and location of the current class's doodles (via PSDoodle).

1: $res := \{\}$ // map: screen → score
2: **for** $doodles$ in $sketch$ **do** // iterate over icon classes
3: $res_{ddl} := \{\}$ // map: screen → score
4: $res_{ddl} := search(doodles)$ // scores for class doodles
5: **for** $screen$ in res_{ddl} **do** // normalize and update
6: $res[screen] += \frac{res_{ddl}[screen]}{max(res_{ddl})} * len(doodles)$
7: **end for**
8: **end for**
9: **for** $text$ in $queries$ **do** // iterate over text queries
10: $res_{txt} := \{\}$ // map: screen → score
11: $res_{txt} := search(text)$ // scores for text
12: **for** $screen$ in res_{txt} **do** // normalize and update
13: $res[screen] += \frac{res_{txt}[screen]}{max(res_{txt})}$
14: **end for**
15: **end for**
16: sort(res) // sort screens by score in descending order

5.4 Evaluation

We evaluated TpD by comparing its top-10 retrieval accuracy, response time, and relevance score with the state-of-the-art approaches PSDoodle and Swire, which are both

sketch-only. Specifically, we explore the following research questions to assess TpD's effectiveness.

RQ1 At similar tool runtime, can PSDoodle achieve similar top-10 accuracy as state-of-the-art screen search approaches?

RQ2 At similar top-10 accuracy, can TpD reduce the state-of-the-art approaches' overall runtime?

RQ3 Does TpD retrieve results relevant to the query consisting of text and UI element sketches?

RQ4 When given a choice, do TpD users search for a screen via a mix of UI element sketches and text or do they exclusively use one of these styles?

To measure how well the various approaches perform, we use a top-k retrieval accuracy metric. Top-k retrieval accuracy is a commonly used metric in information retrieval (including by all of our competitor approaches). Top-k retrieval accuracy is also a good indicator of user satisfaction [46]. To measure this accuracy, we randomly selected 26 of the 58k Rico screens that (a) have at least two TpD-supported UI elements and (b) have a corresponding Swire drawing. We then showed a participant such a screen as a target screen and measured how high TpD ranked this target screen in its search results.

To evaluate TpD, we recruited 10 Computer Science students for a user study. These participants had a similar background and qualification as the user study participants conducted to assess the effectiveness of PSDoodle [14]. Specifically, we selected participants with no formal UI/UX design training. To ensure a diverse group, we recruited five participants with prior mobile app development experience (3 female, 2 male) and five without prior mobile app development experience (2 female, 3 male). Each participant received USD 10 as compensation for their time. For this study, they accessed TpD for the first time via the internet on their own device.

During the study, each participant spent on average 10 minutes going through an interactive tutorial on a website hosted on `http://pixeltoapp.com//toolInsVT/`. The tutorial covered the basics of TpD's visual query language for sketching UI elements, drawing techniques, and search functionalities of TpD. We then instructed each participant to search for at least three screens, one screen at a time.

The website recorded the participant's drawings, text queries, the results of their search query, and the total time they spent for each target screen (which includes time for sketching, waiting for tool feedback, and thinking about how to proceed with a search). We observed the participants' performance via screen sharing but did not further guide or influence the participants.

The instructions directed a participant to keep searching until they found the target screen in the search results or until three minutes passed and we enforced this time limit. Once a participant completed the search for a target screen, we asked the participants to rate how relevant they felt the top 10 search results were to their query. All this information is available in the TpD repository.

5.4.1 RQ1: Achieving state-of-the-art performance in Top-10 Accuracy

We asked 10 participants to search for 3 target screens each, yielding 30 search sessions. We measured the top-10 retrieval accuracy and found that out of the 30 final queries, TpD displayed the target screen in the top-10 search results 27 times, resulting in a 90% top-10 accuracy. This achievement exceeds the top-10 accuracy of PSDoodle (88.2%), Swire (61%), and matches the performance of its subsequent study (90.1%).

5.4.2 RQ2: Efficient and Interactive Screen Retrieval

We analyzed the time participants searched for a target screen using TpD and compared it with the state-of-the-art tools Swire and PSDoodle. TpD search sessions on average

Table 5.2. A participant(P) iteratively issued s sketch and x text queries and spent t[s] time(in seconds) to retrieve a specified target screen using TpD. For most of the session, the target screen's position (p) was in the top 10. The participant (P) also provided ratings indicating the number of top-10 relevant search results (r) to the query.

P	Target 1					Target 2					Target 3				
	s	x	t[s]	p	r	s	x	t[s]	p	r	s	x	t[s]	p	r
1	3	1	96	4	8	2	1	32	4	6	3	2	71	1	4
2	2	3	12	1	4	4	1	50	29	0	2	3	156	1	2
3	6	0	89	15	6	2	1	23	1	5	3	0	52	2	4
4	3	1	125	1	4	3	1	38	2	8	3	0	19	11	10
5	2	1	15	3	6	2	1	5	3	2	3	2	60	1	6
6	2	1	26	2	3	3	0	38	4	10	2	1	12	7	4
7	2	1	41	1	10	2	1	17	4	7	2	2	18	5	5
8	2	4	90	1	5	3	2	14	7	6	5	0	57	1	5
9	3	2	48	2	3	2	1	28	2	7	2	1	17	4	4
10	2	1	30	1	4	3	0	68	3	4	3	1	18	1	4

took less time than the other two tools. The average time it took for Swire participants to draw a Rico screen was 246s, while the duration of sketching per search session using PS-Doodle ranged from 30s to 259s, with an average of 107s. In comparison, the search time for TpD varied from 5s to 156s with an average of 45s. Table 5.2 provides the complete duration of each search session for all 30 experiments, including the final position of the target Rico screen in the search results.

TpD maintains PSDoodle's iterative search approach by allowing users to refine their search queries by adding or removing text or sketches. The search process for TpD begins with adding the first text or drawing. Swire requires users to sketch every element on a mobile screen, resulting in the same number of UI elements as a Rico mobile screen. On average, Swire sketches of the 26 target screens had 21.1 UI elements. For query sessions of PSDoodle, the participants sketched an average of 5.5 UI elements. In TpD, the number of text and UI element sketches in a single session was only 3.9. These results suggest that TpD can find the target screen more accurately with less information than its competitors.

Additionally, TpD offers interactive search functionality and is hosted on AWS. We observed during our experiments that TpD updated the top-10 result screens on the user's website in less than 2 seconds from the time the user submitted a search query. Apart from communication with AWS, the primary time factors were sketch recognition (below 0.1 seconds), screen similarity calculation, and screen ranking (below 1 second).

5.4.3 RQ3: Displaying Multiple Relevant Screens

Participants evaluated the quality of TpD's top-10 result screens after each of 30 search sessions, resulting in the evaluation of a total of 300 screens. Among these, participants judged 156 screens as relevant to their search query. Table 5.2 displays the study results, indicating that TpD successfully retrieved multiple relevant screens for each query. The relevance score achieved by TpD is 52%, which surpassed the score of 42% obtained by PSDoodle.

As an example of relevant TpD results, each row in the Figure 5.1 motivating example depicts a query and TpD's top 5 search results. Most search results accurately position the sketched UI element, and most have text queries on the screen. Overall, the results suggest that TpD is better at retrieving multiple relevant search results.

5.4.4 RQ4: Improving Search Relevance via Text & Sketch

While TpD surpassed PSDoodle in terms of accuracy and speed, another key advantage of TpD is that it consistently presented the target screen on the first page of 50 search result screens, requiring users to provide fewer query information. In contrast, during PSDoodle's evaluation, 2 out of 34 participants failed to find the target screen on the first page within 3 minutes of their query session.

Many of PSDoodle's shortcomings are due to inaccurate clustering of UI elements in Liu's work. TpD can side-step this inaccurate clustering of UI elements, by enabling users

to pinpoint the screen via text queries. As illustrated in the first row of Figure 5.7, TpD ranks the target screen within the top 900 but fails to bring the target screen to the first page using only sketch queries due to the misclassification of the “switch” icon. However, by adding a single text query, the participant obtained the desired result within the top 30. This observation highlights the utility of incorporating text queries in conjunction with sketch queries to improve the accuracy and efficiency of the system in retrieving relevant results.

Table 5.2 illustrates how users combined text and sketches to retrieve the desired Rico screen. In 30 search sessions, 25 combined text and sketches, while the remaining five solely used drawings.

5.4.5 Screen Searches Ranked Outside Top-10

Notably, on three occasions, TpD could not locate the target screen within the top 10 search results. Figure 5.7 depicts two out of the three search sessions. In one instance, During the second session, Participant-2 searched the target screen (right) by sketching “close” and three “switch” icons. Initially, the target screen was ranked relatively low, around 900, due to Rico’s hierarchy labeling the “switch” icon as “check-in” for the target screen. However, after the user added a text query ”t:display”, it was ranked in the top 30.

In another instance, Participant-3 searched for the target screen by sketching a “play” icon, but TpD failed to fetch the target screen because Liu et al. labeled the “play” icon as an image. However, when Participant-3 added more search terms including “menu”, “image”, “share”, “squiggle”, and “search”, TpD was able to re-rank the target screen as rank 13,900, 16,425, 3,162, 88, and 15.

In both cases, the participants stopped searching as the target screen appeared on the first page of the search results. The results obtained from the study suggest that TpD can reliably fetch a target screen with high accuracy. All these findings are available in the TpD repository.

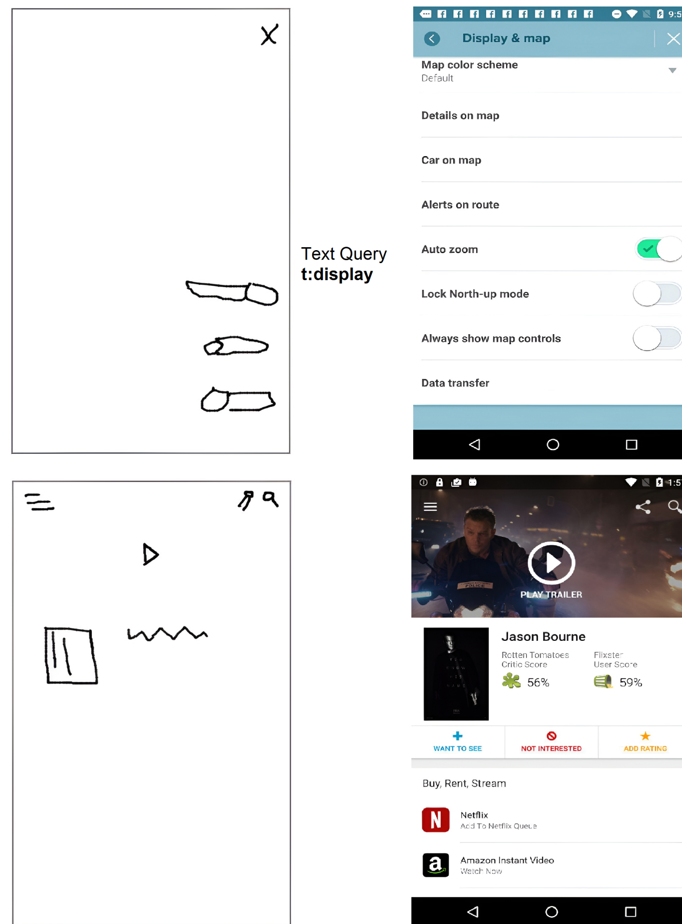


Figure 5.7. During TpD evaluation study, there were three sessions where the target screen did not appear in the top 10 results. In one case, Participant-2 searched for the target screen (top-right) by sketching 'close,' three 'switch' icons, and text query 't: display.' In another case, Participant-3 searched the target screen (bottom-right) by sketching 'menu,' 'play,' 'image,' 'share,' 'squiggle,' and 'search.' The participant stops the search when TpD brings the target screen to the first page of results (top 50)..

5.4.6 User Feedback

Eight of ten participants have responded to our post-experiment survey. The survey contained the following three open-ended questions.

Q1 How can we improve your search interface?

Q2 What was your overall experience with the search results?

Q3 How was your overall experience?

For the first question, users have provided suggestions to improve interactivity and make minor changes to the user interface. For example, one participant asked us "[to] enhance the UI for the website". Another participant left us with the following notable quote.

> "Refine the Green buttons, iterate through a 'not so comic' UI. eg, Green next button."

In response to the second question, participants provided feedback regarding the search results and expressed satisfaction with the current output. Following quotes from participants are noteworthy.

> "They were surprisingly accurate quite quickly, but I don't know how big the data set is."
>
> "The idea and the accuracy really amazed me."
>
> "Great, bit of lag with the refresh, but that's ok."

Lastly, participants rated their overall experience with the tool on a scale of 1 to 10. Responses ranged from 6 to 10. The users' average and median ratings were 8. These results and the survey's open-ended responses are accessible in TpD's repository.

5.5 Conclusions

The process of locating a particular mobile app screen in existing repositories is conventionally restricted to keyword searches or necessitates the creation of either a complete sketch, as seen in Swire, or a partial sketch, as seen in PSDoodle. TpD is the first approach that combines text and drawing input to search for mobile screens, offering an interactive and iterative search engine. TpD developed using a combination of the Rico repository, which contains approximately 58k Android app screens, the Google QuickDraw dataset comprising icon-level doodles, and DoodleUINet curated corpus of some 10,000 app icon

doodles. In our assessment involving software developers external to our team, TpD outperformed existing interactive searching tools by providing superior top-10 search accuracy with considerably less time required to complete the search. Thus, TpD constitutes a significant advancement in the mobile app screen search field, and it is available for use by interested parties under open-source licenses.

CHAPTER 6

D2S2: Drag 'n' Drop Mobile App Screen Search

6.1 Introduction

Iterative app screen search, while an exciting area of recent work [14, 15, 17], still faces several challenges. First, Google image search is fast, searches many images from the open web, and supports arbitrary text-based search queries. But searching for an app screen via a text query remains clumsy, especially when looking for screens that contain certain UI elements in specific locations. Such searches result in long convoluted text queries and produce surprisingly few relevant results. Recent work has searched the smaller widely-used Rico dataset via a combination of text search [17] and sketched element doodles [14, 15]. These approaches support a small number of UI element types via deep learning. Expanding their scope would thus require additional specialized training datasets, which must be collected and curated.

In designing mobile applications, including real-world examples proves advantageous as it aids in gathering requirements, comprehending current trends, and cultivating motivation to develop a compelling mobile app [36, 37]. Given the broad and rapidly expanding market for mobile app development, an efficient mobile screen search tool becomes valuable to keep up with the demand.

Designers commonly use drag-and-drop tools, such as Figma[1], to create wireframes. Similarly, software developers utilize drag-and-drop-based visual kits, like the Android Layout Editor[2], and Prototypr[3], for UI development purposes. The popularity of these

[1]https://www.figma.com/

[2]https://developer.android.com/studio/write/layout-editor

[3]https://prototypr.io/

techniques is growing due to their user-friendly nature, intuitive interfaces, and the fact that they do not require specialized technical expertise [82]. The D2S2 offers an interactive solution based on drag and drop functionality for mobile screen search.

D2S2 is for novice users who need help creating a complete UI design during the early stages of software development. Users can search for mobile screens by dragging and dropping the UI elements onto the canvas. The tool's search interface includes essential features such as undo and redo for user interaction. Users can also add plain text and put text in a text-button. As a user adds, removes, resizes, and moves UI elements, D2S2 searches through 58k Rico [16] screens to fetch UI examples based on UI element type, position, element shape, and texts as shown in Figure 6.1. D2S2 fetches the top-20 screens and displays them in its website's top-pick screen search results section.

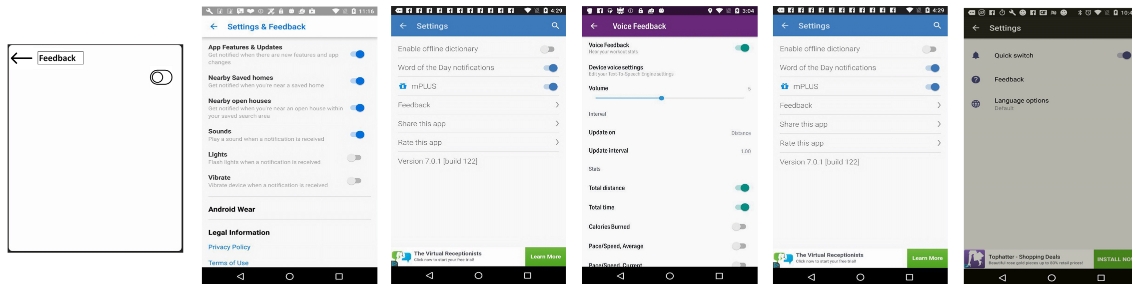


Figure 6.1. Sample D2S2 search query consisting of 3 UI elements (left); searching through 58k mobile app screens, D2S2's top-five search result screens (right) all contain the query's UI elements at about the locations where they appear in the query screen. .

We recruited ten software developers without prior UI/UX design training to assess D2S2's effectiveness. The participants used D2S2 to search for a given target Rico screen until the screen appeared in D2S2's top-20 search results. In our experiment, D2S2 successfully obtained 15/19 target screens within a minute and 19/19 within four minutes. D2S2 further retrieved more relevant mobile screens for user queries than the other closely re-

lated competitor, Google image search. In summary, this paper makes the following major contributions.

- D2S2 is the first interactive drag-and-drop app screen search tool. After each query change it updates its search results.
- D2S2 searches 58k Android screens and is freely available (**`http://pixeltoapp.com/D2S2`**).
- In a preliminary user study, D2S2 performed similarly as the deep-learning based TpD (but without requiring training) and better than Google image search.
- D2S2's implementation (`https://github.com/toni-tang/D2S2`) is available under a permissive open-source license.

6.2 Background

D2S2 searches 58k mobile Android app screenshots from the Rico dataset by Deka et al. [16]. Each screenshot has a corresponding DOM-tree container hierarchy, where each UI element is described by its Android class name, x/y coordinates, textual information, and on-screen visibility. Liu et al. expanded on this dataset by collecting 73k screen elements, categorizing them into 25 types of UI components, and further dividing text buttons into 197 and icon into 135 sub-classes [42]. D2S2 incorporates several common Android UI elements identified by Liu et al.

Previous studies have explored using sketches and wireframe images to search for relevant mobile screens. However, wireframe-based approaches such as Swire rely on complete wireframe images to identify screens with similar visual characteristics, often not considering UI element type and text within the screen [83, 51]. Dependence on an entire wireframe image does not support the iterative nature of the design process.

Our close competitors are PSDoodle [14, 15] and TpD [17], which offer an interactive and iterative approach to searching mobile screens. PSDoodle employs a deep neural network to identify sketched UI elements and then computes a ranking score for Rico's screens based on various factors, including UI element type, position, and shape. TpD extends PSDoodle by adding a text-based search that matches a text query with visible text on the mobile screen and UI element descriptions. Notably, TpD allows queries to contain text, UI element sketches, or both.

6.3 Overview and Design

To create a search system that is easy to use, we followed a user-centric approach. Via the Figma[4] graphical design tool, we thus first created a UI prototype (Figure 6.2), showed the prototype to 11 computer science undergraduate students, and collected their feedback. By incorporating their feedback we then iteratively enhanced D2S2's user experience, mostly by refining D2S2's UI. All user feedback is in D2S2's repository.

6.3.1 User Interface & Query Language

In D2S2, a search query consists of a set of UI elements arranged on a canvas that models the screenshot of a mobile app. Starting with an empty canvas, the user interactively refines this canvas, adding and adjusting UI elements as they should appear on the desired app screens. Each time the user modifies this search query, D2S2 retrieves matching app screens that have the query's UI elements at about the location the user placed them on the canvas. A part of the search is matching any texts the user added to the search query with screens' text contents and descriptions of their UI elements.

Figure 6.3 shows D2S2's current UI. Besides moving the app bar to the bottom, the biggest change is allowing users to search D2S2's library of 52 built-in UI elements by the

[4] https://www.figma.com/

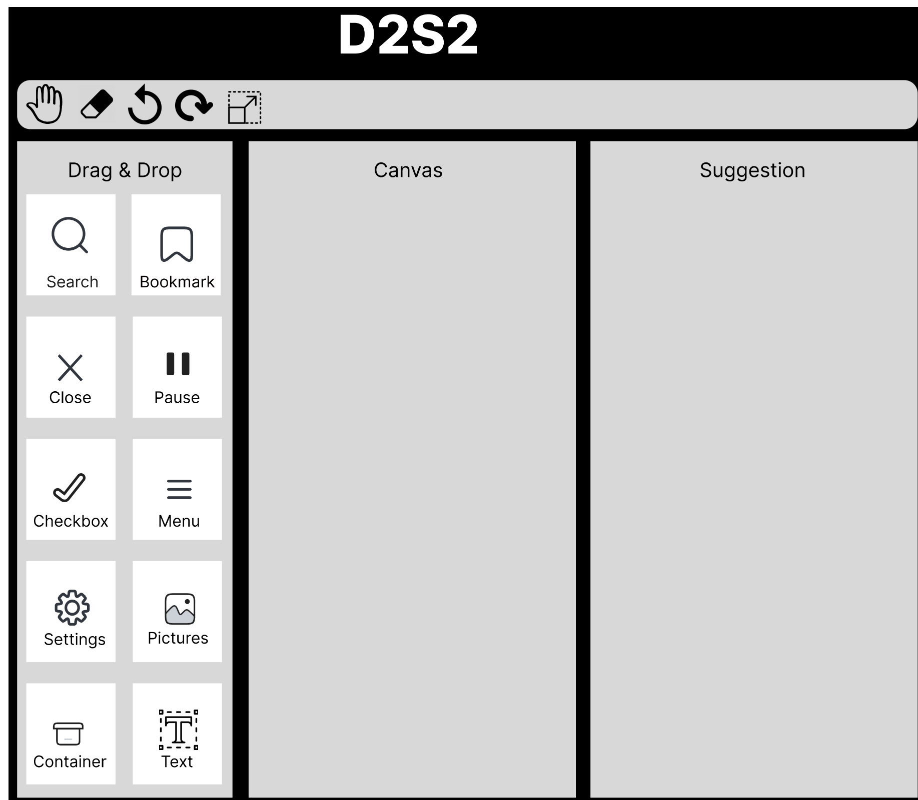


Figure 6.2. Initial D2S2 UI mock-up..

UI element's name and various synonyms. Figure 6.4 lists these 52 UI elements in the order D2S2' UI presents them. The order is TpD's UI elements first, then UI elements identified by Liu et al., ordered by how common they appear in Rico screens [42].

Liu et al. classified the UI elements of Rico's screens into 25 categories. 11/25 categories are various container types, which D2S2 represents via a single general container. D2S2 directly supports 11/14 of the remaining categories, plus 44/135 icon types. Combining 3/56 of these UI elements due to their similar visual representation with another UI element (e.g., slider vs. slider icon) yields D2S2's 52 UI element types plus text.

The user searches (or scrolls) the UI element list, selects a UI element and drags and drops it on the canvas. The user can interact with the UI element, i.e., to move or resize it there. The app bar at the bottom of the canvas allows undoing and redoing the last

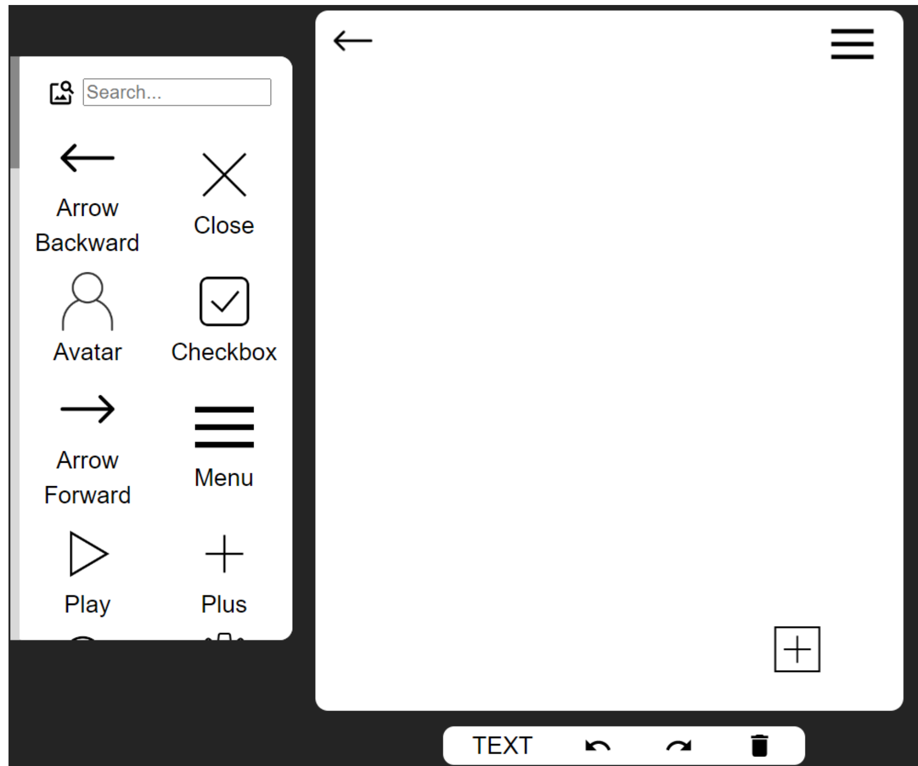


Figure 6.3. D2S2's webpage: Searchable UI elements (left), canvas with 3 user-placed UI elements (right), app bar (bottom). Cropped screenshot (from `http://http://pixeltoapp.com/D2S2`), not shown: D2S2's top-20 search result screen gallery. .

element modifications and clearing the screen. The user can add text either via a text-button from the UI element collection or via the app bar's "TEXT" feature. The latter adds a text element to the canvas the user can manipulate like any other UI element. Clicking on such a text element enables modifying its text content.

As in the earlier PSDoodle and TpD, UI elements may be nested, i.e., to support UI elements grouped in a container element. D2S2 encodes the canvas's current state as a set of 6-tuples of the form (x, y, w, h, c, t), one tuple per UI element on the canvas. The tuple lists an element's left-top corner's location in pixel-space, the UI element's width, height, category, and text content (for text and text buttons). The D2S2 webpage is written

in React, as it provides client-side rendering [84] and efficiently manages various events such as drag-start, drag-end, and the undo/redo functionality.

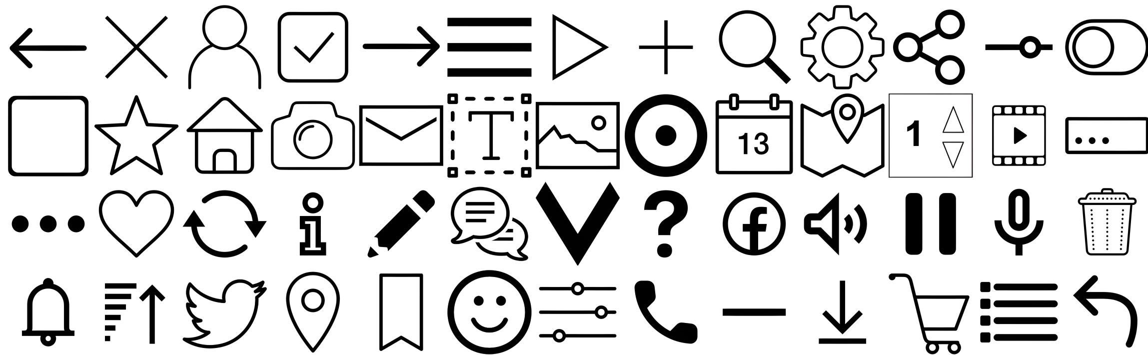


Figure 6.4. A user builds an app screen search query on D2S2's canvas by dragging and adjusting text or these 52 UI elements. .

6.3.2 D2S2's Back-end

Figure 6.5 illustrates D2S2's overall architecture, which consists of D2S2's webpage front-end and its AWS-hosted back-end. Each time the user modifies the query, D2S2 sends the updated query's tuple encoding via HTTP post request to AWS EC2. For the current query, the D2S2 back-end ranks its 58k Rico screens and sends the IDs of the top-20 ranked screens back to the front end. The front-end then retrieves a lower-resolution version of the 20 corresponding screen images from D2S2's AWS S3 bucket and displays them in the top-pick gallery. When clicking on a result screen, D2S2 fetches and displays a higher-resolution version of the result screen.

To rank its 58k screens, D2S2 utilizes the TpD infrastructure, which in turn builds on the PSDoodle infrastructure. For non-text UI elements, D2S2 uses the screen scoring scheme described by PSDoodle (which TpD similarly reused). Specifically, D2S2 divides a mobile app screen into 24 equally sized tiles (6 along the width and 4 along the height)

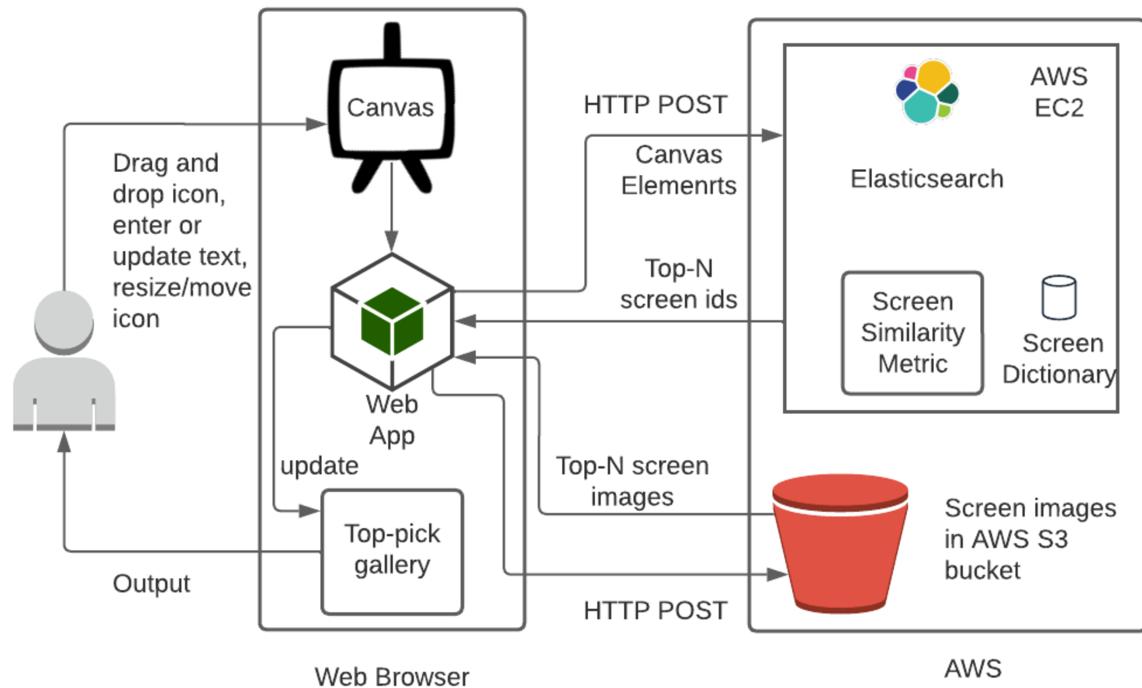


Figure 6.5. D2S2's architecture: A user drags or adjusts Figure 6.4 UI elements on the Figure 6.3 D2S2 front-end webpage (left), which communicates with its AWS-hosted back-end (right)..

and maintains TpD's tile configuration. The main change is in more than doubling TpD's 23 UI element classes to D2S2's 52. In the back-end this is straight-forward, by adding one screen ID index for each of the additional UI element classes to allow fast screen lookup.

For text elements, D2S2 reuses TpD's pipeline [17], which pre-processes the Rico screens' text contents and UI element descriptions (remove stop words, identify names, lemmatization, adding synonyms via contextual analysis, and tagging text content with on-screen location). As for text contents TpD only supports four different screen areas (top-left, top-right, bottom-left, and bottom-right), D2S2 first maps the location of a text element to one of these four TpD screen areas. As TpD, D2S2 then uses ElasticSearch with Levenshtein edit distance one, to heuristically also match slightly mis-typed user-provided text to screen contents.

6.4 D2S2 usage

To compare with its closest competitors TpD and Google Image Search, we enlisted 10 computer science students who did not have formal UI/UX design training. While the participants are different, we recruited them using the same criteria the TpD study used. To ensure diversity across participants, the recruitment selected five individuals with and five without previous mobile app development experience. All participants were early-stage undergraduates aged 20–25. As a token of appreciation, each participant received USD 10 compensation. Specifically, we are interested in the following research questions.

RQ1 How does D2S2 compare with TpD, in terms of total time of the interactive search, final queries' UI element counts, and final queries' top-k screen retrieval accuracy?

RQ2 How does D2S2 compare with Google image search on a free user query, in terms of producing relevant top-20 search results?

For each participant, we had one video conference meeting of about 30 minutes that started with us explaining D2S2's objectives. We then demonstrated the search process for an icon, dragging the icon to the canvas, resizing and adjusting the icon's position on the canvas, the functionality of the undo/redo/clear-screen buttons, and how to add text using the text and text-button features. Each participant accessed D2S2 over the internet via a web browser on their personal machine. We used D2S2's standard setup as a website hosted on an Amazon AWS EC2 general-purpose instance (t2.large), featuring two virtual CPUs and 8 GB of RAM. D2S2's repository contains all experimental results.

6.4.1 Similar Screen Search Performance as TpD

For this second part of a participant meeting, we used the 26 randomly selected Rico target screens used by TpD's evaluation. For each participant, we randomly selected from this pool one target screen per search session. We instructed the participant to create a query

that would retrieve the target screen and refine the query until the target screens appeared in D2S2's top-20 results.

Table 6.1. Participants' search sessions for target screens via D2S2 (left) and free search via D2S2 and Google Images (right): t = search session's total time; n = final query's UI elements (including texts); r = target screen's rank for final query; G/D = top-20 Google/D2S2 results participant judged relevant.

	Target 1			**Target 2**			**Free 1**		**Free 2**	
	t[s]	n	r	t[s]	n	r	G	D	G	D
1	40	4	2	27	3	1	2	18	18	0
2	120	9	1	52	3	2	1	3	3	4
3	37	3	4	48	4	3	5	20	0	20
4	50	3	10	23	2	4	8	16	10	20
5	70	4	1	240	2	1	0	7	4	13
6	59	3	12	196	15	14	3	16	4	20
7	63	3	18	51	3	2	7	11	1	3
8	42	4	16	51	3	7	7	10	0	7
9	39	3	9	42	7	17	1	10	1	19
10	50	4	8	-	-	-	2	5	-	-

Nine participants used D2S2 twice, and one used it once, yielding 19 D2S2 search sessions. For each such search session, we recorded the total time, the number of UI elements and texts in the participant's final query, and the target screen's rank in D2S2's results for that final query. D2S2's top-k retrieval accuracy is the number of search sessions in which D2S2 ranks the target screen in its answer to the participant's final query in the top-k. We use top-k retrieval accuracy, as the metric is widely used to evaluate related work [85, 86, 34] and correlates with user satisfaction [46].

Table 6.1 summarizes the results. Comparing these results with TpD's results is a little tricky as TpD's participants were instructed to search until the target screen appears in TpD's top-10 search results or the search exceeds 3 minutes. So TpD participants were encouraged to spend a bit of additional time maybe to refine a query. With this caveat, the

overall results for D2S2 and TpD are similar. D2S2's top-20 screen retrieval accuracy is 100% (19/19) vs. TpD's 97% (29/30).

D2S2's total search session time was at least 23 seconds, 240s maximum, 68s average, and 50s median. This compares to a 5s minimum, 156s maximum, average 45s, and 35s median for TpD. Contributing to TpD shorter search sessions are TpD's experimental setup (which allowed participants to practice using TpD for some 10 minutes before collecting results) and D2S2 having more than twice the number of UI elements to choose from for a search query. We observed participants using significant time browsing the UI elements available in D2S2 and selecting the correct UI element.

6.4.2 More Targeted Than Google Image Search

In this final part of a participant meeting, we instructed each participant to formulate a Google-style search query and perform a corresponding search using both D2S2 and Google image search (an example of a participant's query is "mobile screen menu icon top left and search icon top right"). We then asked the participant to rate each result in both tools' top-20 results as relevant or non-relevant to the participant's query.

Participants judged 20% (77/380) of Google image search's results as relevant and 58% (222/380) of D2S2's result screens. D2S2's 58% relevance here is largely in line with TpD's 52% reported for searches for a given target screen [17]. Given D2S2's and TpD's slightly different experimental setups, it is hard to draw conclusions about their relative performance. For the search scenario over 58k Rico screens, both tools clearly perform better than Google image search.

6.5 Conclusions

Current sketch-based iterative searching of mobile screens has limitations in supporting many UI elements. In light of this, drag-and-drop-based solutions provide a flexible

alternative. D2S2' provides an interactive, drag-and-drop search experience that displays search results in real-time. The tool is freely available and has undergone user testing, demonstrating its effectiveness. In summary, D2S2 presents a promising solution for novice users who require assistance creating a comprehensive UI design during the initial phases of a mobile application.

CHAPTER 7

Conclusions

In summary, this dissertation presents innovative approaches for incorporating low-fidelity sketches into software development, resulting in the creation of several tools for software engineers. These tools use sketching to enhance mobile application development. DoodleUINet addressed the need for publicly available sketches of common UI element categories at the stroke-sequence level. Our tools, Doodle2App showcase the potential of drawings in generating ready-to-deploy mobile apps that can alleviate the burden of repetitive and low-level technical details associated with app development. PSDoodle presents an interactive and iterative approach for acquiring real-world mobile screens using low-fidelity sketches. The TpD incorporates a combination of keyword-based searches and partial drawings, resulting in a more efficient solution for mobile screen retrieval. Additionally, we have developed an alternative drag-and-drop method for searching mobile screens. The results obtained from the user evaluation demonstrate that these tools hold promise in assisting inexperienced software developers in mobile application development.

REFERENCES

[1] J. A. Landay and B. A. Myers, “Interactive sketching for the early stages of user interface design,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, May 1995, pp. 43–50.

[2] M. W. Newman and J. A. Landay, “Sitemaps, storyboards, and specifications: A sketch of Web site design practice as manifested through artifacts,” EECS Department, UC Berkeley, Tech. Rep. UCB/CSD-99-1062, 1999.

[3] P. F. Campos and N. J. Nunes, “Practitioner tools and workstyles for user-interface design,” *IEEE Software*, vol. 24, no. 1, pp. 73–80, Jan. 2007.

[4] G. Ines, S. Makram, C. Mabrouka, and A. Mourad, “Evaluation of mobile interfaces as an optimization problem,” *Procedia computer science*, vol. 112, pp. 235–248, 2017.

[5] T. D. Hellmann and F. Maurer, “Rule-based exploratory testing of graphical user interfaces,” in *2011 Agile Conference*. IEEE, 2011, pp. 107–116.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. 26th Annual Conference on Neural Information Processing Systems (NIPS)*. NIPS, Dec. 2012, pp. 1106–1114.

[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *arXiv:1409.1556*, 2015.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[9] M. Andersson, A. Maja, and S. Hedar, “Sketch classification with neural networks: A comparative study of CNN and RNN on the Quick, Draw! data set,” Master’s thesis, Uppsala University, June 2018.

[10] D. Ha and D. Eck, “A neural representation of sketch drawings,” 2017.

[11] J. Jongejan, H. Rowley, T. Kawashima, J. Kim, and N. Fox-Gieg, “Quick, draw!” 2016, accessed March 2022. [Online]. Available: https://quickdraw.withgoogle.com/

[12] S. Mohian and C. Csallner, “DoodleUINet: Repository for DoodleUINet Drawings Dataset and Scripts,” July 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5144472

[13] S. Mohian and C. Csallner, “Doodle2App: Native app code by freehand UI sketching,” in *Proc. 7th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), Tool Demos and Mobile Apps Track*. ACM, May 2020, pp. 81–84.

[14] S. Mohian and C. Csallner, “Psdoodle: fast app screen search via partial screen doodle,” in *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2022, pp. 89–99.

[15] S. Mohian and C. Csallner, “Psdoodle: searching for app screens via interactive sketching,” in *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2022, pp. 84–88.

[16] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, “Rico: A mobile app dataset for building data-driven design applications,” in *Proc. 30th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2017, pp. 845–854.

[17] S. Mohian and C. Csallner, “Searching mobile app screens via text + doodle,” 2023.

[18] Y. Y. Wong, "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, 1992, pp. 83–84.

[19] P. Campos and N. J. Nunes, "Practitioner tools and workstyles for user-interface design," *IEEE software*, vol. 24, no. 1, pp. 73–80, Jan. 2007.

[20] A. S. Carter and C. D. Hundhausen, "How is user interface prototyping really done in practice? a survey of user interface designers," in *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Sept. 2010, pp. 207–211.

[21] J. A. Landay and B. A. Myers, "Sketching interfaces: Toward more human interface design," *IEEE Computer*, vol. 34, no. 3, pp. 56–64, Mar. 2001.

[22] D. Fichou, "Teleport vision api v2," July 2019, march 2022. [Online]. Available: https://teleporthq.io/blog-new-vision-api

[23] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge, "JavaSketchIt: Issues in sketching the look of user interfaces," in *Proc. AAAI Spring Symposium on Sketch Understanding*. AAAI, Mar. 2002, pp. 9–14.

[24] A. Coyette, S. Kieffer, and J. Vanderdonckt, "Multi-fidelity prototyping of user interfaces," in *Proc. INTERACT*. Springer, Sept. 2007, pp. 150–164.

[25] M. de Sà, L. Carriço, L. Duarte, and T. Reis, "A mixed-fidelity prototyping tool for mobile devices," in *Proc. AVI*. ACM, May 2008, pp. 225–232.

[26] J. Seifert *et al.*, "Mobidev: A tool for creating apps on mobile phones," in *Proc. Mobile HCI*. ACM, Aug. 2011, pp. 109–112.

[27] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui," in *Proc. ASE*. IEEE, Nov. 2015, pp. 248–259.

[28] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 2015.

[29] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[30] P. Sangkloy, N. Burnell, C. Ham, and J. Hays, “The Sketchy database: Learning to retrieve badly drawn bunnies,” *ACM Transactions on Graphics*, vol. 35, no. 4, pp. 119:1–119:12, July 2016.

[31] F. Li, R. Fergus, and P. Perona, “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories,” in *Proc. CVPRW*. IEEE, June 2004.

[32] L. Torrey and J. Shavlik, “Transfer learning,” in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, Aug. 2009, pp. 242–264.

[33] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proc. OSDI*. USENIX, Nov. 2016, pp. 265–283.

[34] F. Huang, J. F. Canny, and J. Nichols, “Swire: Sketch-based user interface retrieval,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, May 2019.

[35] A. Sain, A. K. Bhunia, Y. Yang, T. Xiang, and Y.-Z. Song, “Cross-modal hierarchical modelling for fine-grained sketch based image retrieval,” in *Proc. 31st British Machine Vision Virtual Conference (BMVC)*, 2020.

[36] S. R. Herring, C.-C. Chang, J. Krantzler, and B. P. Bailey, “Getting inspired! understanding how and why examples are used in creative design practice,” in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2009, pp. 87–96.

[37] C. Eckert and M. Stacey, “Sources of inspiration: a language of design,” *Design studies*, vol. 21, no. 5, pp. 523–538, 2000.

[38] D. Ritchie, A. A. Kejriwal, and S. R. Klemmer, "d. tour: Style-based exploration of design example galleries," in *Proc. 24th annual ACM Symposium on User Interface Software and Technology (UIST)*, 2011, pp. 165–174.

[39] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 2009, pp. 183–192.

[40] S. Mohian and C. Csallner, "soumikmohianuta/PSDoodle: PSDoodle Repository for the Publication," Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6339717

[41] B. Deka, Z. Huang, and R. Kumar, "ERICA: interaction mining mobile apps," in *Proc. 29th Annual Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2016, pp. 767–776.

[42] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *Proc. 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2018, pp. 569–579.

[43] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[44] Tensorflow, "Recurrent neural networks for drawing classification," Dec 2020. [Online]. Available: https://github.com/tensorflow/docs/blob/master/site/en/r1/tutorials/sequences/recurrent_quickdraw.md

[45] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[46] S. B. Huffman and M. Hochster, "How well does result relevance predict session satisfaction?" in *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, July 2007, pp. 567–574.

[47] A. Chalechale, G. Naghdy, and A. Mertins, "Sketch-based image matching using angular partitioning," *IEEE Transactions on Systems, Man, and Cybernetics-part a: systems and humans*, vol. 35, no. 1, pp. 28–41, 2004.

[48] R. Hu and J. Collomosse, "A performance evaluation of gradient field hog descriptor for sketch based image retrieval," *Computer Vision and Image Understanding*, vol. 117, no. 7, pp. 790–806, 2013.

[49] J. Song, Q. Yu, Y.-Z. Song, T. Xiang, and T. M. Hospedales, "Deep spatial-semantic attention for fine-grained sketch-based image retrieval," in *Proc. IEEE International Conference on Computer Vision*, 2017, pp. 5551–5560.

[50] S. K. Yelamarthi, S. K. Reddy, A. Mishra, and A. Mittal, "A zero-shot framework for sketch based image retrieval," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 300–317.

[51] S. Bunian, K. Li, C. Jemmali, C. Harteveld, Y. Fu, and M. S. Seif El-Nasr, "Vins: Visual search for mobile user interface design," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–14.

[52] B. Adefris, "Sketch2code," Jun 2020. [Online]. Available: https://www.kaggle.com/biniamad/sketch2code

[53] V. P. Sermuga Pandian, S. Suleri, and P. D. M. Jarke, "Uisketch: A large-scale dataset of ui element sketches," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–14.

[54] B. Lee, S. Srivastava, R. Kumar, R. Brafman, and S. R. Klemmer, "Designing with interactive example galleries," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2010, pp. 2257–2266.

[55] C. Bernal-Cárdenas, K. Moran, M. Tufano, Z. Liu, L. Nan, Z. Shi, and D. Poshyvanyk, "Guigle: A gui search engine for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 71–74.

[56] K. Kolthoff, C. Bartelt, and S. P. Ponzetto, "Data-driven prototyping via natural-language-based gui retrieval," *Automated Software Engineering*, vol. 30, no. 1, p. 13, 2023.

[57] "User interface (ui) design market size, share, growth, and industry analysis, by type(user experience (ux) design, interaction design (id), visual & graphic design and others), by application(software and app, web page, game, tv interfaces and others), regional forecast to 2028)." [Online]. Available: https://www.businessresearchinsights.com/market-reports/user-interface-ui-design-market-102500

[58] "Google quickdraw." [Online]. Available: https://quickdraw.withgoogle.com/

[59] B. Elasticsearch, "Elasticsearch," *Internet: https://www. elastic.co,[Nov. 21, 2022]*, 2018.

[60] C. Manning and H. Schutze, *Foundations of statistical natural language processing*. MIT press, 1999.

[61] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[62] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[63] J. B. Lovins, "Development of a stemming algorithm." *Mech. Transl. Comput. Linguistics*, vol. 11, no. 1-2, pp. 22–31, 1968.

[64] M. F. Porter, "An algorithm for suffix stripping," *Program*, 1980.

[65] V. Balakrishnan and E. Lloyd-Yemoh, "Stemming and lemmatization: A comparison of retrieval performances," 2014.

[66] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength natural language processing (NLP) in Python," 2020.

[67] explosion_ai, "Release en_core_web_trf-3.4.1 · explosion/spacy-models." [Online]. Available: https://github.com/explosion/spacy-models/releases/tag/en_core_web_trf-3.4.1

[68] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *Acm Computing Surveys (CSUR)*, vol. 44, no. 1, pp. 1–50, 2012.

[69] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[70] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[71] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[72] E. M. Dharma, F. L. Gaol, H. Leslie, H. Warnars, and B. Soewito, "The accuracy comparison among word2vec, glove, and fasttext towards convolution neural network (cnn) text classification," *J Theor Appl Inf Technol*, vol. 100, no. 2, p. 31, 2022.

[73] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[74] C. Fellbaum, "Wordnet and wordnets," 2005.

[75] R. Speer, J. Chin, and C. Havasi, "Conceptnet 5.5: An open multilingual graph of general knowledge," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[76] M.-H. Hsu, M.-F. Tsai, and H.-H. Chen, "Combining wordnet and conceptnet for automatic query expansion: A learning approach," in *Information Retrieval Technology: 4th Asia Infomation Retrieval Symposium, AIRS 2008, Harbin, China, January 15-18, 2008 Revised Selected Papers 4*. Springer, 2008, pp. 213–224.

[77] H. Fang, "A re-examination of query expansion using lexical resources," in *proceedings of ACL-08: HLT*, 2008, pp. 139–147.

[78] M.-H. Hsu, M.-F. Tsai, and H.-H. Chen, "Query expansion with ConceptNet and WordNet: An intrinsic comparison," in *Information Retrieval Technology: Third Asia Information Retrieval Symposium, AIRS 2006, Singapore, October 16-18, 2006. Proceedings 3*. Springer, 2006, pp. 1–13.

[79] M. S. Divya and S. K. Goyal, "An advanced and quick search technique to handle voluminous data," *Compusoft*, vol. 2, pp. 171–175, 2013.

[80] V.-A. Zamfir, M. Carabas, C. Carabas, and N. Tapus, "Systems monitoring and big data analysis using the elasticsearch system," in *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. IEEE, 2019, pp. 188–193.

[81] A. M. Namboodiri and A. K. Jain, "Online handwritten script recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 1, pp. 124–130, 2004.

[82] J. Joy, "Review on different types of drag and drop mobile app development platforms," 2018.

[83] J. Chen, C. Chen, Z. Xing, X. Xia, L. Zhu, J. Grundy, and J. Wang, "Wireframe-based ui design search through image autoencoder," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–31, 2020.

[84] M. F. S. Lazuardy and D. Anggraini, "Modern front end web architectures with react. js and next. js," *Research Journal of Advanced Engineering and Science*, vol. 7, no. 1, pp. 132–141, 2022.

[85] S. Dey, P. Riba, A. Dutta, J. Llados, and Y.-Z. Song, "Doodle to search: Practical zero-shot sketch-based image retrieval," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2179–2188.

[86] J. Collomosse, T. Bui, and H. Jin, "Livesketch: Query perturbations for guided sketch-based visual search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2879–2887.