Doodle2App: Native App Code by Freehand UI Sketching

Soumik Mohian
soumik.mohian@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Christoph Csallner
csallner@uta.edu
University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

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 collections have enabled online sketch detection that supports many sketch element classes at high classification accuracy. This paper leverages the recent Google Quick, Draw! dataset of 50M sketch stroke sequences to pre-train a recurrent neural network and retrains it with sketch stroke sequences we collected via Amazon Mechanical Turk. The resulting Doodle2App website offers a paper substitute, i.e., a drawing interface with interactive UI preview and can convert sketches to a compilable single-page Android application. On 712 sketch samples Doodle2App achieved higher accuracy than the state-of-the-art tool Teleport. A video demo is at https://youtu.be/P4sb0pKTNEY

CCS CONCEPTS

• **Software and its engineering** → *Software prototyping*; • **Human-centered computing** → **User interface toolkits**.

KEYWORDS

User interface design, sketching, prototyping, GUI, deep learning

ACM Reference Format:
Soumik Mohian and Christoph Csallner. 2020. Doodle2App: Native App Code by Freehand UI Sketching. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387905.3388607

1 INTRODUCTION

User interface development of many apps starts with freehand sketching, typically with pen on paper [3, 4, 12, 17, 23]. 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 [8, 13]. Some only support a few sketch primitives (e.g., ellipse, rectangle, straight line, and squiggly line) and thus support few user-app interaction styles [2, 5, 6, 12, 13, 21]. The others have low recognition accuracy [8].

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 [4]. 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! ("Quick-Draw") collection of over 50M labeled sketches of 345 categories, from "aircraft carrier" to "zigzag", each given as a stroke sequence [9]. 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

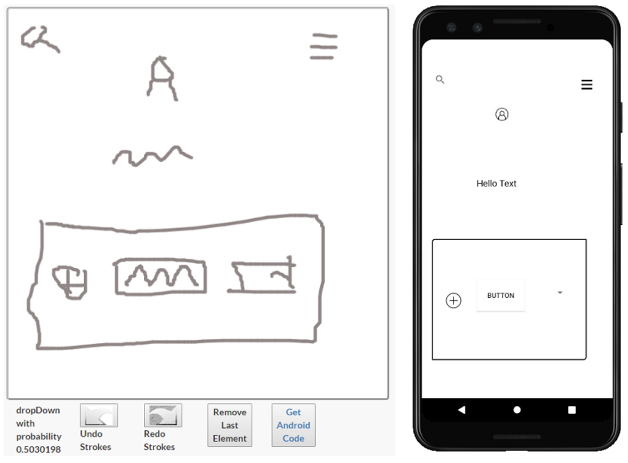

Figure 1: Drawing interface and generated Android app.

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 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 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 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 [18] or Teleport [8]). In contrast, an online approach processes a sketch's strokes in the order they are drawn (e.g., as SILK [13]). 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.

2.1 Sketch to Code with Teleport

The most closely related approach is Teleport's vision API v2 [8]. It supports 21 classes of hand-drawn UI element sketches. Doodle2App's atomic UI element classes (Figure 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 Recurrent Neural Networks (RNNs)

State-of-the-art approaches for image recognition typically use deep learning and especially convolutional neural networks (CNNs) [11]. 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 [15]. Doodle2App builds on QuickDraw's network architecture to leverage its recent sketch recognition success [9]. QuickDraw uses bi-directional RNNs [20], which use both stroke sequences and reverse stroke sequences.

3 UI ELEMENTS & SKETCH SAMPLES

Doodle2App currently focuses on Android. The Rico dataset [7] 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**.

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). Besides primitives, Doodle2App also supports an example compound element:

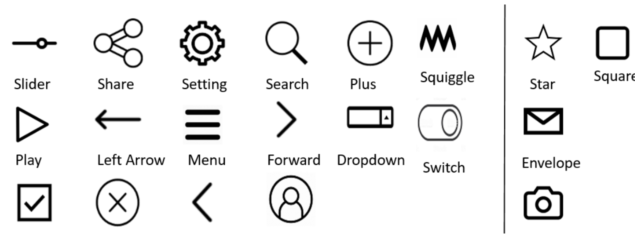

Figure 2: The 20 graphical primitives Doodle2App currently recognizes. Samples for 4 classes (right) are from QuickDraw, the rest (left) is from Mechanical Turk.

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.

The QuickDraw dataset [9] contains 345 sketch categories (from "aircraft carrier" to "zigzag"), with some 100k samples each, drawn by anonymous users [10]. 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 we built a website similar to Sketchy [19]. 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.

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 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 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.

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 [14], we used transfer learning [22] 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 [1], trained the existing QuickDraw network architecture [9] 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 [16].

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 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.

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.

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.

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%.

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 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.

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

ACKNOWLEDGMENTS

Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with UTA policy. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1527398 and 1911017.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. OSDI.* USENIX, 265–283.
[2] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. 2002. JavaSketchIt: Issues in sketching the look of user interfaces. In *Proc. AAAI Spring Symposium on Sketch Understanding.* AAAI, 9–14.
[3] Pedro Campos and Nuno Jardim Nunes. 2007. Practitioner tools and workstyles for user-interface design. *IEEE software* 24, 1 (Jan. 2007), 73–80.
[4] Adam S. Carter and Christopher D. Hundhausen. 2010. How is user interface prototyping really done in practice? A survey of user interface designers. In *Proc. VL/HCC.* IEEE, 207–211.
[5] Adrien Coyette, Suzanne Kieffer, and Jean Vanderdonckt. 2007. Multi-fidelity prototyping of user interfaces. In *Proc. INTERACT.* Springer, 150–164.
[6] Marco de Sà, Luís Carriço, Luís Duarte, and Tiago Reis. 2008. A mixed-fidelity prototyping tool for mobile devices. In *Proc. AVI.* ACM, 225–232.
[7] Biplab Deka et al. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proc. UIST.* ACM, 845–854.
[8] Dimitri Fichou. 2019. Teleport Vision API v2. https://teleporthq.io/blog-new-vision-api Accessed March 2020.
[9] David Ha and Douglas Eck. 2018. A neural representation of sketch drawings. In *Proc. ICLR.* OpenReview.net.
[10] Jonas Jongejan, Henry Rowley, Takashi Kawashima, Jongmin Kim, and Nick Fox-Gieg. 2016. Quick, Draw! https://quickdraw.withgoogle.com/ Accessed March 2020.
[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proc. NIPS.* NIPS, 1106–1114.
[12] James A. Landay and Brad A. Myers. 1995. Interactive sketching for the early stages of user interface design. In *Proc. CHI.* ACM, 43–50.
[13] James A. Landay and Brad A. Myers. 2001. Sketching interfaces: Toward more human interface design. *IEEE Computer* 34, 3 (March 2001), 56–64.
[14] Fei-Fei Li, Rob Fergus, and Pietro Perona. 2004. Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. In *Proc. CVPRW.* IEEE.
[15] Zachary C Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. (2015). arXiv:1506.00019
[16] Soumik Mohian and Christoph Csallner. 2020. *Repository for DoodleUINet Drawings Dataset and Scripts.* https://doi.org/10.5281/zenodo.3653552
[17] Mark W. Newman and James A. Landay. 1999. *Sitemaps, storyboards, and specifications: A sketch of Web site design practice as manifested through artifacts.* Technical Report UCB/CSD-99-1062. EECS Department, UC Berkeley.
[18] Tuan A. Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI. In *Proc. ASE.* IEEE, 248–259.
[19] Patsorn Sangkloy, Nathan Burnell, Cusuh Ham, and James Hays. 2016. The Sketchy database: Learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics* 35, 4 (July 2016), 119:1–119:12.
[20] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.
[21] Julian Seifert et al. 2011. Mobidev: A tool for creating apps on mobile phones. In *Proc. Mobile HCI.* ACM, 109–112.
[22] Lisa Torrey and Jude Shavlik. 2009. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques.* IGI Global, 242–264.
[23] Yin Yin Wong. 1992. Rough and ready prototypes: Lessons from graphic design. In *Proc. CHI, Posters and Short Talks.* ACM, 83–84.