Rewire: Interface Design Assistance from Examples

Amanda Swearngin[1,2], **Mira Dontcheva**[1], **Wilmot Li**[1], **Joel Brandt**[1,*], **Morgan Dixon**[1,**], **Andrew J. Ko**[3]

Adobe Research[1]
Seattle, WA, 98103
{mirad,wilmotli}@adobe.com

Paul G. Allen School[2]
University of Washington
Seattle, WA, 98195
amaswea@cs.washington.edu

The Information School[3]
University of Washington
Seattle, WA, 98195
ajko@uw.edu

ABSTRACT

Interface designers often use screenshot images of example designs as building blocks for new designs. Since images are unstructured and hard to edit, designers typically reconstruct screenshots with vector graphics tools in order to reuse or edit parts of the design. Unfortunately, this reconstruction process is tedious and slow. We present *Rewire*, an interactive system that helps designers leverage example screenshots. Rewire automatically infers a vector representation of screenshots where each UI component is a separate object with editable shape and style properties. Based on this representation, the system provides three design assistance modes that help designers reuse or redraw components of the example design. The results from our quantitative and user evaluations demonstrate that Rewire can generate accurate vector representations of interface screenshots found in the wild and that design assistance enables users to reconstruct and edit example designs more efficiently compared to a baseline design tool. .

ACM Classification Keywords

H.5.2. Information Interfaces: User Interfaces—*prototyping*

Author Keywords

User interface design; wireframing; pixel-based reverse engineering.

INTRODUCTION

Examples play a critical role in the interface design process. Designers browse and curate example galleries for inspiration [13, 5], explore alternatives with examples [13, 11], and use examples as building blocks when developing new designs [16]. One common requirement across many of these scenarios is the need to reuse or edit parts of an example design. For instance, a designer may want to edit the colors of UI components to explore different palettes, change text labels based on the target application, or reuse part of an example interface in a new design. Performing such operations requires example designs to be represented in an editable way, predominantly as vector graphics where interface components are specified as objects with properties that define their shape and appearance.

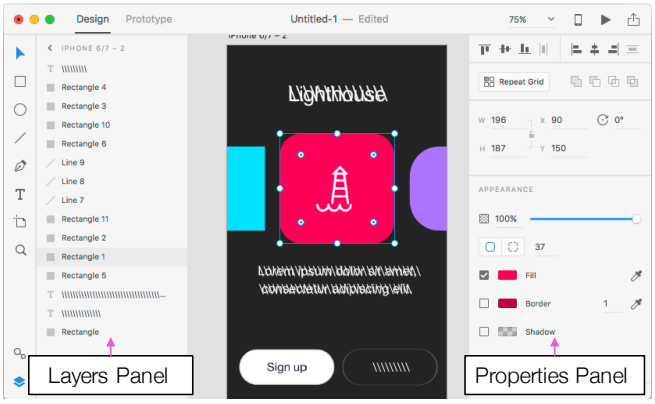

Figure 1. Rewire's Full Vector design assistance mode, in the Adobe Experience Design (XD) canvas. Designers activate the mode by right-clicking on a screenshot. Designers can then edit the properties and layering of the vectorized output in XD's Properties and Layers panels.

While vector representations enable reuse and editing, designers can't easily collect examples in this form because interfaces are often composed of images and shapes that combine in unexpected ways. Designers may also find examples in real interfaces where they cannot access a vector representation. Thus designers typically collect example designs in the form of screenshots (i.e., raster images). Most devices have shortcuts to copy portions of the screen to an image, which makes it easy for designers to capture examples. At the same time, unlike vector graphics, screenshots are flat, unstructured, and hard to edit. As a result, when a designer wants to modify a design from an image, they need to reconstruct all or relevant parts of the content to produce an editable vector representation. This representation must be created by hand by drawing and specifying the properties of shapes through trial-and-error.

Designers can use commercial vectorization tools, like Illustrator's ImageTrace, to ease this process. However, because these tools aim for visual fidelity, they represent their output with path objects representing visually distinct boundaries in an image. With paths, changing properties like rectangle corner radius requires editing many individual control points to modify the relevant boundaries. Changing font properties of text represented as a path is impossible without creating a text box. Thus, we aim to extract higher-level semantic objects (e.g., rectangles, text boxes) with properties of common UI components (e.g., corner radius, font type). This paper explores how

*Joel is at Snap, Inc. (jbrandt@snap.com).
**Morgan is at Cheeseburger Therapy (morgan.dixon@gmail.com).

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

ISBN 978-1-4503-5620-6/18/04…15.00

DOI: https://doi.org/10.1145/3173574.3174078

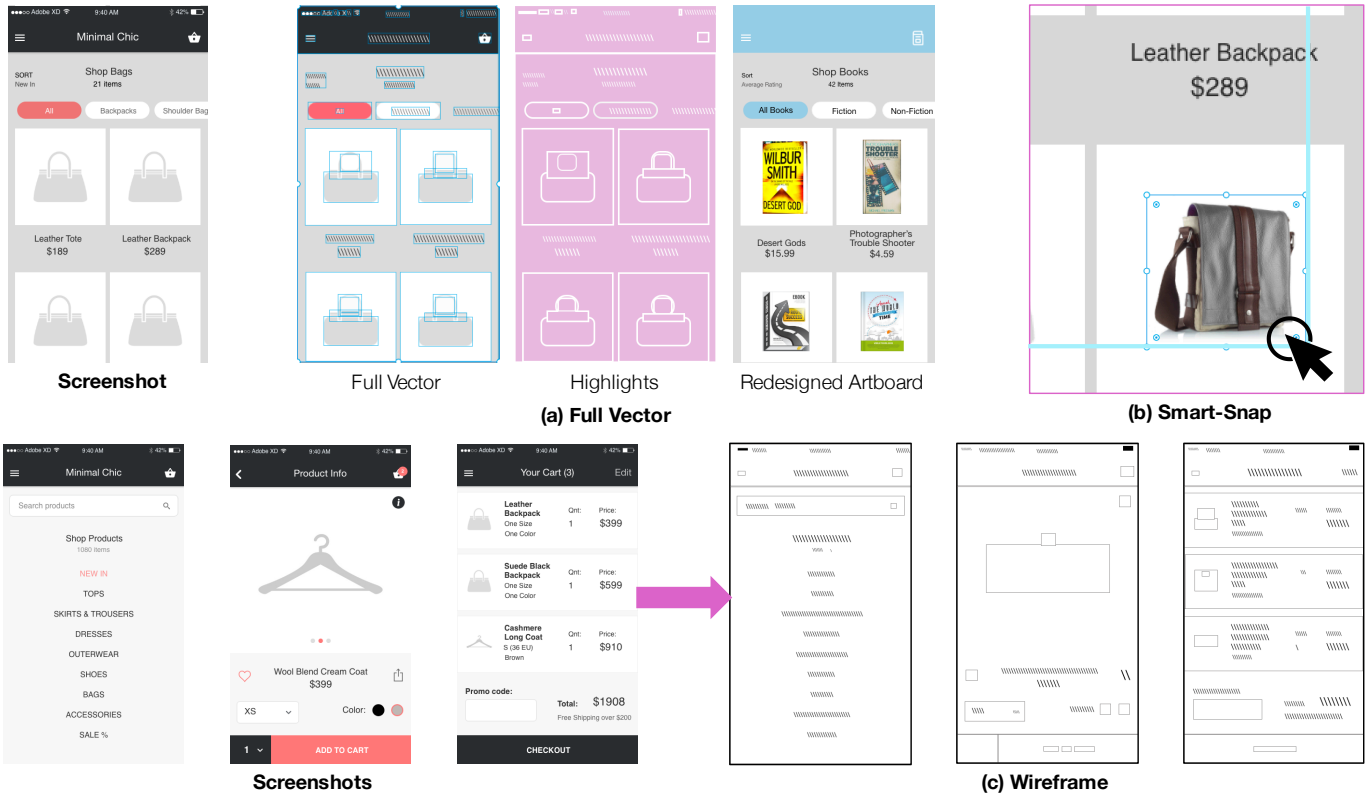

Figure 2. Rewire provides three modes of design assistance. The first mode, Full Vector (a), creates vector objects for shapes in the image. Designers can highlight the automatically vectorized shapes by toggling the pink Highlights layer. Designers can then update and redesign the vectorized artboard, as shown on the right. The second mode, Smart-Snap (b), displays alignment and spacing guidelines to help designers align newly drawn shapes to shapes in the screenshot. The third mode, Wireframe (c), generates abstract wireframes of the screenshot, removing most visual details.

to achieve this goal using image analysis techniques to uncover semantic structure and properties interface screenshots.

Our system, *Rewire*, helps designers reuse and edit example screenshots. Rewire infers a semantic vector representation from a screenshot and provides three design assistance modes from these inferred vectors. *Full Vector*, shown in Figure 1 and 2 (a), creates an artboard with a set of vector shapes inferred from a screenshot. This mode allows designers to quickly reconstruct or edit the example design. *Smart-Snap*, shown in Figure 2 (b), provides snapping guides that become active when drawing new shapes. These guides can assist designers in aligning new UI shapes with those in the example design. *Wireframe*, shown in Figure 2 (c), displays the inferred vector shapes with a simple black outline. This mode helps designers reuse the layout of the example design while abstracting away visual details.

Our work on Rewire includes three main contributions:

1. An automatic method for extracting semantic vector objects from screenshots that combines low-level image processing with UI-specific reverse engineering techniques.
2. Three new design assistance modes that leverage the extracted vector objects to help designers create new designs.
3. Quantitative and qualitative evaluations demonstrating the accuracy of our pipeline and the benefits of our design assistance modes for reconstructing example designs.

FORMATIVE INTERVIEWS & EXAMPLE SCENARIO

We conducted formative interviews with 10 user interface designers working at both small and large companies to help us uncover common use cases for Rewire. All designers frequently used screenshots in their work. In one extreme case, one designer recreated an entire legacy interface design to use as a template, spending days drawing a complex vectorized design document from a screenshot. However, most designers said that they mostly recreate only parts of a design they need to change for making quick mockups. Sometimes they will simply cut, paste, and resize parts of screenshots into their designs for quick prototyping, and then recreate UI shapes when moving to high-fidelity. Designers also mentioned recreating from screenshots when the original design assets were lost and when clients sent them interface screenshots to incorporate into their designs.

From these observed use cases, we developed three design assistance modes that help designers leverage interface screenshots. We describe and motivate these modes in the context of an example scenario. In the scenario, Maria, a user experience designer, performs several design tasks using an existing vector-based design tool, Adobe Experience Design (XD), that has been augmented with the Rewire design assistance modes.

Maria is creating a mockup for a shopping cart page. Her project manager sends her the screenshot shown in Figure 2 and asks for a similar design with realistic bag images in place

of the grey bag icons. To accomplish the task, Maria opens the screenshot in XD and activates Rewire's Smart-Snap (Figure 2, (b)) mode. As she drags an image of a leather purse onto the canvas, blue snapping guides visualize how the image aligns with the interface shapes in the example design. These guides enable Maria to quickly align and resize four realistic bag images over the original bag icons, without having to carefully manipulate the size and position of each image.

Maria's next assignment is to modify the same shopping cart page to show a set of sample books rather than bags, with book titles and prices below each item. The project manager wants Maria to create several variations of the design with different color schemes. Instead of redrawing, retyping, and matching the properties of the screenshot by hand, Maria activates the Full Vector mode to automatically generate vectorized objects from the screenshot (Figure 2 (a), Full Vector). To see the generated objects, Maria enables Rewire's highlighting feature as shown in Figure 2 ((a), Highlights). Rewire renders shapes (e.g., rectangles, circles, lines) in pink with white outlines, and text objects with white backslashes to indicate they are editable. Maria edits the text for each book and modifies the header and button colors to create several design variations. In this setting, the Full Vector mode helps Maria to quickly create designs using the components in the original screenshot.

Finally, Maria's manager sends her several screenshots of inspirational examples and asks her to show the client a range of potential designs based on these images. Since the goal is to present high-level ideas, Maria wants to show abstracted versions of the example designs that leave out unnecessary (and possibly distracting) design details like the specific fonts or icons. Maria drags the screenshots into XD and activates Rewire's Wireframe mode, shown in Figure 2 (c), to automatically create wireframe representations. Rewire draws these with a simple black outline with no additional styling. Maria then labels the components of the wireframe to highlight key parts of the app such as the header and shopping cart items.

RELATED WORK

Our work is inspired by prior research on data-driven design. Much of this work explores how examples can be used for design inspiration [16, 14, 7] or retargeted to different styles and layouts [15]. We advance this body of work by proposing novel design assistance modes that help users leverage examples when creating new designs. Moreover, while many existing techniques analyze design examples with explicit structure (e.g., the DOM in web pages), our approach automatically infers useful structure from flat screenshots of user interfaces, which are convenient for designers to collect.

Our method for analyzing screenshots integrates Prefab [8, 10], which reverse engineers user interface structure from images to modify those interfaces at run-time. Two other related systems are Sikuli Script [25], which automatically drives user interfaces based on their appearance, and Remaui [18], which generates Android application code from example images. In contrast to this previous work, we apply structure extraction techniques to aid the design (rather than control) of user interfaces, which imposes specific requirements on the analysis. In particular, Prefab, Sikuli and Remaui all aim to identify specific widget types (e.g., buttons, checkboxes) for automation or code generation, while Rewire detects primitive shapes (e.g., rectangles, circles) and extracts their visual properties to produce an editable, vectorized version of the design. Finally, while most UI automation systems require users to select training examples of the relevant widgets, Rewire's screenshot processing pipeline does not require designers to label any example interface elements.

Deep learning offers a potential approach to extract structure from screenshots. The pix2code system [4] presents a convolutional and recurrent neural network to infer interface code from screenshot images. The network represents the desired structure via a constrained domain-specific language (DSL) that encodes simple geometric relationships between a fixed set of UI components in addition to a small set of style properties, like color. Yet, this DSL is not designed to handle the much broader range of component types, appearances and arrangements that arise in many example screenshots. While adapting this network to output the type of structured representation that Rewire requires is an interesting direction for future work, the pix2code approach in its current form is not directly applicable to our problem.

Finally, as part of our screenshot processing pipeline, Rewire performs layout beautification to improve the inter-component alignment and visual consistency of the vectorized output. Xu et. al. introduce the term layout beautification for refining the alignment of sketch-based interfaces [24] and propose interactive techniques for visualizing, modifying and enforcing potential layout constraints. The work of O'Donovan et al. [19, 20] uses an energy-based model to rate the quality of layouts based on design principles and applies machine learning to synthesize layouts for single page graphic designs. Inspired by these methods, we adopt a constrained optimization approach to satisfy alignment, distribution and consistency constraints.

REWIRE ARCHITECTURE

We assume the input to Rewire to be an image of an interface. Because interfaces consist of an array of geometric shapes and natural images, contain complex hierarchies, and contain a large set of properties that frequently interact, we focused on detecting and vectorizing four primitive shape types: rectangles, circles, lines, and text. These shapes can be combined or used individually to represent most interface elements. The output of Rewire's processing pipeline is a vectorized artboard containing editable shapes, as shown in Figure 3 (Output). These shapes contain styling (e.g. corner radius) and size properties. Rewire populates these shapes into a vector-drawing tool, where designers can edit, resize, or move them.

To support the *Smart-Snap*, *Full Vector* and *Wireframe* design assistance modes, each vectorized shape is presented as demonstrated in Figure 2. The screenshot processing pipeline consists of three stages, as illustrated in Figure 3:

Segmentation. First, we segment the screenshot into regions that represent distinct geometric elements by leveraging existing low-level computer vision algorithms (Figure 3, Stage 1). We also classify segments into a predefined set of primitives (e.g. rectangles, lines, circles, text).

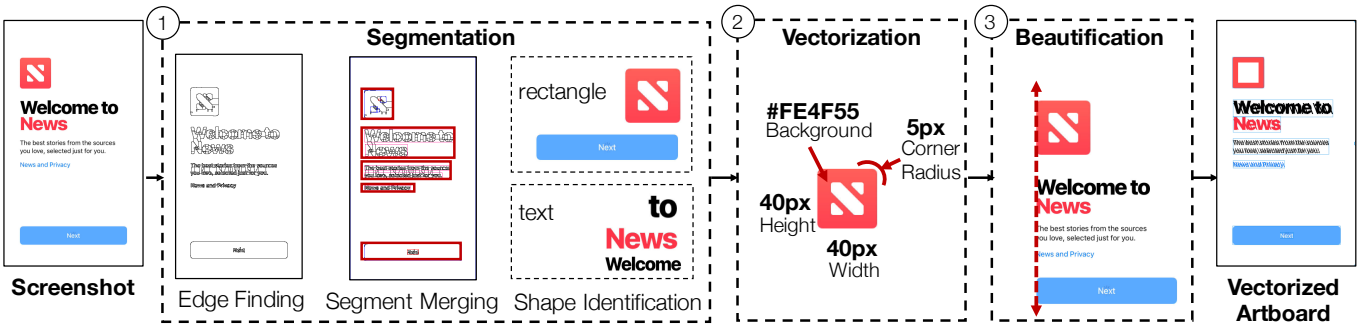

Figure 3. System overview of Rewire. The system input is a screenshot. Rewire segments shapes from the image and classifies them by primitive shape type (1), extracts properties of segments to create vector shapes (2), and beautifies (i.e., aligns & normalizes) the resulting layout (3).

Vectorization. For each segment, we generate a corresponding vector object by estimating the relevant shape (e.g., size, position) and style (e.g., color, border thickness) properties (Figure 3, Stage 2).

Beautification. Finally, we refine the properties of individual vector objects via a global optimization that tries to improve alignment and consistency across the entire artboard (Figure 3, Stage 3).

Each design assistance mode uses the resulting vectorized output in different ways. The *Full Vector* mode presents the fully vectorized artboard to the user. The *Wireframe* mode removes all the extracted style properties and only preserves the shape of each object. Finally, the *Smart-Snap* mode creates snapping guidelines based on the bounding box of each object but does not add the actual objects to the artboard. We implemented all of the design assistance modes as extensions to Adobe XD.

Segmentation

Rewire extracts primitive shapes from the input screenshot in two steps. First, we detect all text regions via OCR, and then we segment the remaining parts of the screenshot. Identifying the text up front improves the quality of the subsequent segmentation because it allows the algorithm to filter out extraneous small segments that often arise within text regions.

For text detection, we use an existing OCR library called Tesseract [22] to obtain bounding boxes that correspond to potential text shapes.To optimize Tesseract's performance, we set the page segmentation mode to *sparse text* which tries to find as much text as possible, in no particular order. With these parameters, Tesseract outputs individual lines of text.

To filter out obvious false positives, we compute two geometric properties, *solidity* and *extent*, for the pixels contained within each text word. Prior work found these to be good at discriminating between text and non-text regions [12]. The system removes any text words with solidity greater than 0.3 and extent > 0.9 based on previous techniques [12, 17] and experimentation with our example dataset. Finally, to create a concise, easily editable set of text segments, we merge adjacent text lines that are close to each other and similar in color using the Golden Ratio, Φ (1/1.618), a typography ratio that relates font size, line height, and line width in an aesthetically pleasing way. We merge lines if the vertical distance between their bounding boxes is less than the Golden Ratio, Φ, times the mean line height and the correlation between their color histograms (measured via Pearson's coefficient) is greater than 95%. We set these thresholds empirically and use them for all of our results and experiments.

To segment the remaining parts of the screenshot, Rewire decomposes the image into an over-complete set of candidate segments and then iteratively merges and classifies segments to obtain a final set of shapes. We compute candidate segments by constructing an Ultra Metric Contour Map (UCM) [1], which uses low-level image features to partition the image into a set of closed regions. Since we have already detected text regions, we remove UCM region boundaries that overlap with any of the extracted text, and use the remaining segments as our initial set of candidates.

Given the candidate segments, Rewire iteratively merges or removes segments until it has attempted to merge all segments. The first step is to put all candidate segments into a working set S. For each segment $s \in S$, we determine whether it is one of the primitive shapes (i.e. circle, rectangle, line) that Rewire handles. To detect rectangles and lines, we count the non-segment pixels within the axis-aligned bounding box of s. If there are no non-segment pixels and the height or width of the smaller dimension is less than 5px, then we classify the s as a line. This threshold of 5px was set by a manual exploration of common patterns in hand-created designs. If s is not a line but the fraction of non-segment pixels is less than 90%, then we classify the it as a rectangle. Finally, if the s is not a line or a rectangle, we compute a circle Hough transform [3] to check whether it is circular.

If the segment s is classified as one of these shapes and its larger bounding box dimension is bigger than 20px, then we remove it from the working set and add it to the final set of segments. Otherwise, we try to merge s with its adjacent segments. If s is a line, we try to merge it with any adjacent co-linear line segments. If s is not a line, we merge it with adjacent segment t if they are a similar size (i.e., neither segment is more than three times larger than the other) or both s and t are small (i.e., have a width or height less than 5px). If s is merged with any segments, we put the resulting segment back in the working set S. If it is not merged with any segments, we

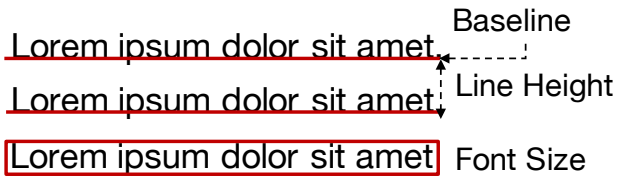

Figure 4. Rewire extracts the baseline, line height, and font size of text shapes.

add it to the final set of segments. Eventually, all segments are removed from the working set.

When the working set is empty, we do one last clean-up pass. We remove any segments that are not primitive shapes and smaller than 25px in area. We discovered through experimentation that these segments frequently correspond to noise produced by the UCM segmentation. For each rectangle, we also remove any lines or nested rectangles that are within 2px of the rectangle boundaries, since these extraneous segments typically correspond to styled rectangle borders. Finally, for every primitive shape, we remove neighboring segments that are less than one tenth the size of the shape because they are likely to be edge segments from border effects or shadowing around the detected segments.

The final output of the segmentation is a set of segments labeled with a primitive shape type. Segments not classified as a primitive shape are left unlabeled.

Vectorization

Rewire generates vector objects from the set of segments. In this phase, text segments become text objects, line segments become lines, circle segments generate circles, and unlabeled and rectangle segments become rectangles. For each segment, the position of the its bounding box determines the position of the corresponding vector object. To ensure shapes are correctly layered, Rewire builds a partial hierarchy of interface shapes based on visual containment. Rewire does not support vectorizing icons or complex vector graphics. Based on formative interviews, there are likely few cases where designers would want to reconstruct an entire logo from a screenshot. Instead, they typically want to abstract it away or replace it with a different icon or logo. Thus, our aim is to reconstruct whole UI components, which can be represented mainly with primitive shapes. Rewire computes the vector properties via the following segment-specific vectorization procedures.

Text

For each text segment, Rewire generates a text object and estimates the baseline, font size and color. For text segments containing more than one line of text, Rewire also estimates the line height property (see Figure 4).

To estimate the baseline, Rewire converts the region of the original image inside the bounding box of the text area into an edge detected image which contains only white and black pixels on edge boundaries. Rewire analyzes a y-coordinate distribution of the white pixels in the region, and sets the baseline as the y-coordinate of the largest group with the highest y-coordinate.

For text areas with more than one line, Rewire estimates the line height property by computing distances between adjacent baselines in a text area, and computing an average between adjacent baselines.

To estimate the font size for text shapes, Rewire uses the bounding box height of the tallest text line in the text area, as shown in Figure 4, which directly converts into a fixed pixel value. Typography defines the font size as the distance between the highest ascender line (i.e., the capital letter L in Figure 4 and the lowest descender line (i.e., the bottom of the p in Figure 4 possible in a line of text. This means that Rewire's font size estimate will be less accurate if the text line does not have ascender and descender lines. To address this, Rewire normalizes font size estimates during the beautification stage (Stage 3 in Figure 3), and snaps font size estimates upward toward similarly sized text shapes in the document.

To extract text color, Rewire first finds the background color of the text by computing a histogram of all pixel colors found at the boundary of the text box, and merging them into groups of indistinguishable colors using the Delta-E metric [21]. The finds the foreground color by clustering pixels in the foreground and computing a weighted average between the number of pixels in a group and the amount of contrast with the background, setting the font color to be the color of the group with the highest weighted average.

Rectangles & Lines

For rectangle segments, Rewire generates a rectangle object and estimates the background color, border color, border thickness, and corner radius. To enable this, we created a new rectangle model in Prefab [8], a system for reverse engineering the pixels of graphical user interfaces. Prefab recognizes widget shapes in an image by fitting the pixels of an image to nine sub-regions: the interior (background), four borders, and four corners. For Rewire, we created a simplified six-region Prefab model that only includes a single corner region. This makes the computation more efficient but restricts the system to generating rectangles with a single fixed corner radius (which is by far the common case).

Prefab's output is the size of each region, and the colors of the longest repeating patterns discovered in the border and background regions. From this, we infer the corner radius (i.e., width/height of the corner region) and border thickness (i.e., width/height of border regions). For background color and border color, if Prefab finds a single solid color, Rewire returns this color. If a solid color is not found, Rewire returns the most common color from this region. If the extracted border regions and background regions have the same color, Rewire collapses them and returns the background color and corner radius, and does not return a border. Note that we do not currently extract gradients or patterned fills for rectangles.

Rewire uses a six-region Prefab model for vertical and horizontal lines but allows the size of the side regions and corner regions to be zero. For lines, Rewire sets the background color and line thickness to the size and color of the background region of the Prefab model.

Circles

For circles, Rewire finds the radius based on the dimensions of the segment bounding box. It extracts the background color by looking for the clustering pixels into foreground and background regions, and selecting the most common background color. Border color and thickness are not currently extracted for circles; however, we believe we could extract these similarly to rectangles using a parametric Prefab model.

Non-text, non-primitive segment

Rewire generates a rectangle shape that aligns with the segment bounding box and clips out the corresponding screenshot pixels. While users cannot easily edit the contents of such clipping regions, they can repurpose them by copying, scaling, and rearranging the clipped pixels. Figure 1 shows a lighthouse icon shape that can be rescaled and moved.

Beautification

The first two stages of the processing pipeline can introduce small misalignments and inaccuracies in the shape properties of generated vector objects. In particular, inexact edge detection (e.g., due to anti-aliasing or border effects) during segmentation can propagate to the vectorized objects. Such inaccuracies impact the quality of the design by creating misalignment between objects or inconsistencies across related elements (e.g., text objects with slightly different font sizes). To address this problem, Rewire adjusts the shape properties of vector objects to improve alignment and consistency across the artboard. We formulate the problem as a constrained optimization with the following soft and hard constraints.

Soft Constraints

We use soft constraints to discourage small differences in the size, alignment, spacing, and text properties of objects. For every pair of non-text vector objects, the system checks whether the widths or heights of the bounding boxes match within a small threshold and if so, adds a constraint penalizing the difference in the relevant dimension. For every pair of non-text objects, we penalize small misalignments between the boundaries (top, bottom, left, right) and centers (horizontal, vertical) of the bounding boxes. Similarly, for every pair of text vector objects, we penalize misalignments along the vertical center and baseline axis, but do not add constraints for the other boundaries because it only makes sense to align text boxes along their baselines. For text/non-text pairs, the constraints are added to align the bottom of the bounding box of the non-text shape to the baseline of the text shape.

In addition, for groups of three or more objects that are approximately aligned along the same axis, we check for potential distribution relationships between the shapes (i.e., when the gaps between adjacent shapes are nearly uniform) and if so, add constraints that penalize discrepancies. Finally, for every pair of text objects, the system penalizes small differences between the baseline, line height or font size. The assumption is that the original designer manually aligned the shapes in the original design, so small misalignments have likely appeared due to small inaccuracies of the screenshot processing pipeline. We use a threshold of 2px to determine when to apply the non-text (size, alignment, spacing) constraints and a 1px threshold for the text constraints.

	# Elements	Rect.	Circ.	Line	Text	Other
Mean	27.5	7.6	1.8	0.9	10.8	0.6
Median	25.5	6	1	0	9.5	0
Min	7	0	0	0	2	0
Max	55	31	10	6	23	6

Table 1. The summary statistics for the total and number of elements of each type per artboard included in our technical evaluation dataset.

Hard Constraints

To prevent the optimization from introducing new artifacts or transforming objects too drastically, we impose two types of hard constraints. First, we set a hard limit on how much any shape or text property can change during the optimization. We found a threshold of 2 pixels to work well. Second, we constrain every object to stay contained within the bounds of its parent object, if it has one. Unlike the soft constraints, these hard constraints are guaranteed not to conflict.

Optimization

We combine these constraints into a cost function which tries to maximize the number of soft constraints satisfied, and minimize the distance the movement of shapes from their original locations. Given the set of soft and hard constraints and the cost function, we use Z3 [6] to obtain a solution. We use the Z3Py library's optimize solver, which enables solving using a cost function and weighted soft constraints.

TECHNICAL EVALUATION

To evaluate the accuracy of our screenshot processing pipeline, and to understand the challenges in of using our system in the wild, we collected a dataset of interface designs in the form of vector drawings, ran our pipeline on screenshot images of each drawing, and compared the vectorized output from Rewire to the ground truth vector representation. Here, we describe our process for creating the ground truth dataset and the evaluation metrics we used to measure Rewire's performance.

Dataset

To obtain a representative collection of user interface designs, we collected vector drawings from popular online design sharing galleries, including dribble.com and designermill.com. We restricted our search to Adobe XD design files so that we could view all the drawings with a single tool. In addition, to keep the dataset self-consistent, we only considered mobile interface designs, which accounted for 39% of the design files. Finally, since Rewire is not yet designed to identify or segment natural images, we filtered out designs with large background images that cover more than 80% of the artboard. We also removed any documents that contained only UI kits, which are large collections of vectorized widgets that typically do not contain any interface designs. Using these rules, we downloaded (on June 20, 2017) a total of 88 XD files containing 203 mobile design drawings across 6 websites.

While these designs were representative, they did not have the appropriate vector structure to use directly as ground truth. Many designs include vector icons or logos that consist of many grouped geometric objects. Since the primary goal of Rewire's pipeline is to reconstruct whole UI components rather than their visual parts, it did not make sense to treat individual

objects within icon or logo groups as part of the ground truth vector representation. However, the naming and granularity of such groups was not consistent, which made it hard to automatically extract the ideal vector structure from each design. In addition, some designs included objects hidden behind other parts of the drawing. Such objects are likely artifacts of the design process designers left behind by accident.

To resolve these issues, we randomly selected a single drawing from each of the 31 XD design files in our collection and manually edited its vector structure. Specifically, we removed any hidden objects and created specially named groups for sets of geometric primitives that form icons or logos. In all cases, icons and logos were easy to identify, and Figure 6 shows some examples (e.g. flower and car). After cleanup, we ended up with a dataset of 31 ground truth vector drawings, with a median of 25 vector objects (see Table 1 for statistics).

Evaluation Method

For the evaluation, we compare Rewire's Full Vector output to the ground truth images. We evaluate the accuracy of SmartSnap or Wireframe modes separately because they use the same object boundaries as the Full Vector output. We compute *precision*, *recall*, and *f-score* for two different evaluation metrics: *type detection* and *property accuracy*.

For *type detection*, we first determine corresponding segments between the Rewire output and ground truth. To measure precision, we consider each Rewire object, compute the standard intersection-over-union (IoU) score for all ground truth objects, and select the one with the highest IoU as the match. If the types of the two matched objects are the same, we count a hit. Otherwise, we count a miss. To measure recall, we perform the inverse procedure starting with each ground truth object. This metric describes the accuracy of Rewire's segmentation and object identification.

To measure the *property accuracy*, we consider each matching pair of Rewire and ground truth objects that have the same type. We check if the objects overlap enough (90% for rectangles and circles, and any amount for lines and text), and for lines we check if they are colinear. For each pair that meets these requirements, test for matching property values using a 2px threshold for pixel-based properties and the Delta-E metric [21] for color similarity with a threshold of 1. We measure property accuracy as the number of properties within this threshold across the artboard, and report these results for Text and Geometry (i.e., rectangles, circles, lines).

Results

Figure 5 shows the distribution of precision, recall, and f-scores for Text and Geometry for the *type detection* and *property accuracy* metrics. The Text Types histogram shows we are able to identify most text shapes, and for both Geometry and Text objects, we successfully extract most properties. For Text Properties, 27/31 artboards have f-scores over 70%, and for Geometry Properties 17/24 artboards have f-scores over 70% accuracy. Note that the Geometry Properties histogram does not include the 7 artboards where Rewire did not match any of the Geometry Types well enough to extract their properties.

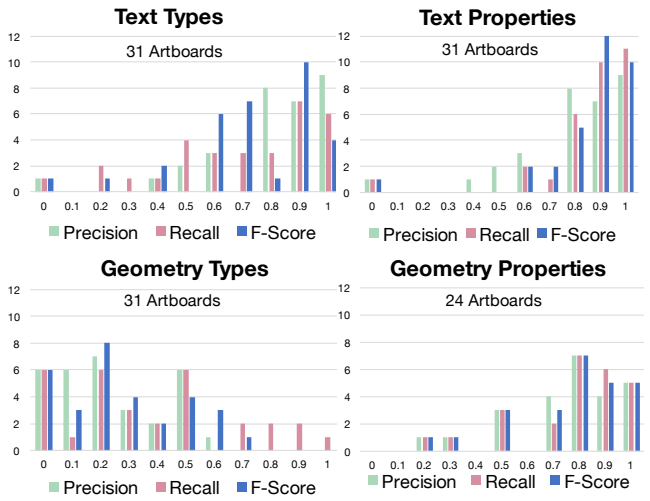

Figure 5. Histograms of Rewire's f-score, precision, and accuracy on the dataset of real artboards collected from popular design sharing galleries. The height of each bar represents the amount of artboards at that accuracy level.

In general, type detection is harder than property extraction because it requires the initial segmentation to be correct. Moreover, identifying the type for Geometry objects is challenging for three key reasons. Natural images result in many extra segments, as shown by the low precision scores in the Geometry Types histogram. If we remove the 12 artboards with natural images from our dataset, 60% of the remaining artboards have Geometry Type f-scores above 50%. Additionally, *small objects* are sometimes mistaken for noise and filtered out by our segmentation. For example, 11 artboards in our dataset have standard mobile header bars with small components. Finally, many designs use *alternate representations* for interface components. For example, designers sometimes use closed paths to draw rectangles and circles, while Rewire treats all geometric shapes as primitives. Layering relationships can also be ambiguous. For the designs in Figure 2, the grey background layer extends beneath the black header rectangle. Rewire extracts two adjacent rectangles. In many cases, Rewire's output may be equivalent in terms of utility and editability.

Extracting more accurate Geometry objects is the biggest opportunity for improvement in our processing pipeline. As we discuss later, there are many directions for future work to address the current limitations. Yet, as we demonstrate in our user study, extracting even a subset of the interface shapes in a screenshot can have practical benefits for design tasks.

USER STUDY

To help us understand the benefits and limitations of Rewire's design assistance modes, we conducted a user study with professional interface designers. We investigated the following research questions:

RQ1: Do Rewire's modes of design assistance improve the *accuracy* and *efficiency* of designers when creating a vector graphics design from an example screenshot?

RQ2: What aspects of each design assistance mode do designers like and dislike?

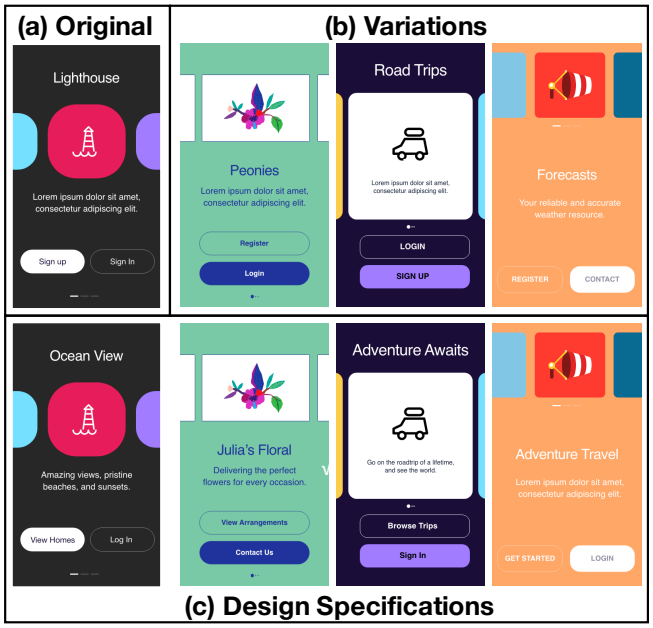

Figure 6. The original screenshot (a), variations (b), and design specifications that designers recreated for our user study (c).

For the study, we recruited 16 former or current user interface designers (6M, 9F) between the ages of 21 and 50, all of whom had at least one year of professional experience. All participants completed four design tasks over the course of 1 hour using a version of Adobe Experience Design (XD) augmented with Rewire's design assistance modes, shown in Figure 1, running on a MacBook Pro (OSX Sierra). While nine participants had little or no previous exposure to XD, they used very similar vector design tools like Bohemian Sketch and Axure in their daily work. The other six designers had at least one year of experience using XD.

Procedure

To evaluate RQ1, we used a within-subjects design with four conditions. Each condition included one task in which we asked participants to produce a vector representation of all of the UI components of an interface (see Figure 6). Rather than asking them to produce an exact vector replica of the screenshot, we made versions of the designs with different text (see Figure 6b) and gave them the target designs on paper. We also gave them printed task instructions for reference. To reduce the overall study time, we told designers to use the default font for all text, even if it didn't match the screenshot. However, we did ask them to match other text properties (e.g., color, size) to preserve the overall appearance and layout of the design. Since we didn't want them to vectorize icons, such as the lighthouse shown in Figure 6 (a), we instructed designers to use screenshots for all icons. Within XD, we provided the input screenshot in one artboard and asked designers to create the new design in an adjacent artboard. To facilitate tracing, we also added the screenshot to the working artboard as a background layer at 30% opacity. We asked participants to complete the task as quickly and accurately as possible and gave them a time limit of seven minutes.

In the *Baseline* condition, the designer used standard XD tools. In the *Smart-Snap* and *Full Vector* conditions, the designers used the corresponding design assistance modes provided by Rewire. Finally, we also measured the performance of an "idealized" version of Rewire, by creating an *Ideal Vector* condition that provided a perfect vector representation of the screenshot. In this last condition, designers only needed to edit the text. Figure 1 shows the XD experience for the Full Vector condition. Designers were able to edit the properties and layering of the auto-generated vectors. The Smart-Snap and Screenshot Only modes started with no shapes in the layers panel, while the Ideal Vector contained the fully vectorized set of shapes. We did not to include the *Wireframe* mode as a test condition because, as noted earlier, designers described this mode as being most useful in communicating with clients.

Before performing any tasks, we asked designers to recreate a small screenshot as a warm-up to familiarize themselves with the task instructions and baseline XD features. After the warm-up, participants performed the tasks under each condition. We fixed the task order and counterbalanced the order of conditions using a Latin square. Before each of the Rewire conditions (*Smart-Snap* and *Full Vector*), we described the relevant features of the design assistance mode and let the user experiment with them on a small screenshot.

To evaluate RQ2, designers completed an open-ended survey after each task about what they liked and disliked about each condition. Additionally, they ranked all conditions in terms of preference and described contexts in which they would find the different modes most useful.

Materials

Since our study design counterbalanced the four conditions across the four tasks, it was important for the tasks to be equivalent in difficulty. We based the four tasks on an example interface from our dataset of online designs. We chose a design with a small number of elements to make the tasks manageable (Figure 6a), and with an average f-score of Rewire performance of 47% which was around the median of results for segmentation from our example dataset.

Using this example as a reference, we then created three variations that had the same number and distribution of object types and rendered the corresponding screenshots (Figure6b). Finally, we verified that Rewire produced similar quality output for all four input images (e.g., the number of incorrect properties or mis-classified objects is comparable).

Design & Analysis: To measure *accuracy*, we exported each of the designer's artboards to a screenshot and computed the pixel difference using the Delta-E metric [21]. We averaged this metric across all pixels for each screenshot, and assign each designer an error score. We selected this metric because we instructed designers to match the original designs as close visually as possible. We did not expect that they should or could recreate the exact shapes and structure of the original artboards. Thus, we manually verified that every participant created shapes as instructed.

To analyze task durations, we first ran a repeated measures Anova to check for significant differences between the con-

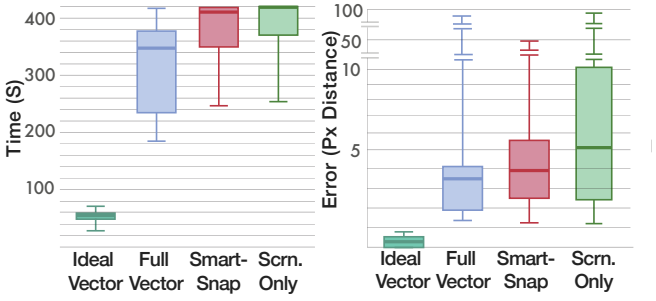

Figure 7. The left shows a box-plot of the the designers' task completion times for Rewire (Smart-Snap and Full Vector) and baseline (Ideal Vector and Screenshot Only) conditions. The right shows amount of error in the designers' output, as measured by the average of pixel color distance.

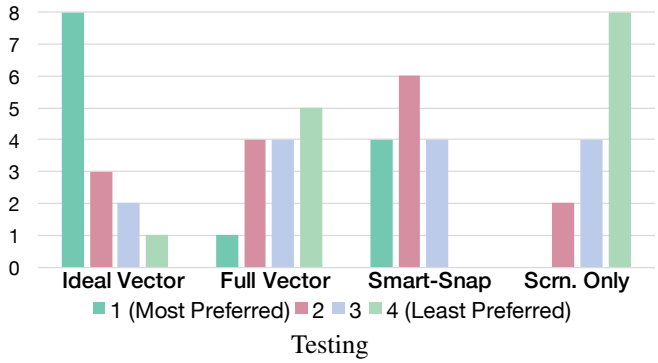

Figure 8. The designers' overall rankings of design assistance mode from most to least preferred.

ditions. Then, we ran pair-wise t-tests (paired two sample for means) across the six pairs of conditions. We then used the Holm-Bonferroni post-hoc method [23] to analyze the significance of the paired conditions and did not reject any null hypothesis where the p-value was greater than this metric. We report the adjusted p-values in our results. We calculated the effect sizes using Cohen's *d*.

Results

RQ1: Speed and Accuracy

Figure 7 shows a box-plot of the designers' times to completion. For the *Full Vector* condition, we found that designers were able to complete the tasks 52 seconds on average faster than the Smart-Snap condition ($t(11) = 3.26$, $p' < 0.008$, $d=0.91$), and 65 seconds faster than the Screenshot Only condition ($t(11) = 4.32$, $p' < 0.002$, $d=1.07$). For *Smart-Snap*, designers completed the tasks 13 seconds faster than the Screenshot Only condition ($t(11) = 2.20$, $p' < 0.025$, $d=0.36$). However, since they still had to recreate all of the shapes, the time savings were not as significant as the Full Vector condition. Additionally, since we stopped designers after 7 minutes, Smart-Snap and Screenshot Only conditions had a significant ceiling effect. For the Full Vector condition, most participants completed the tasks within the allotted time. Thus, the differences may have been more dramatic had there been no ceiling effect. The *Ideal Vector* was 5.5 minutes faster than Smart-Snap ($t(11) = 23.12$, $p' < 2.81E\text{-}10$, $d=7.01$), 4.67 minutes faster than Full Vector ($t(11) = 13.20$, $p' < 8.70E\text{-}08$, $d=1.07$), and 5.7 minutes faster than Screenshot Only ($t(11) = 28.13$, $p' < 4.02E\text{-}11$, $d=8.22$). This demonstrates that we can find significant time improvements by being able to produce a perfectly vectorized output, motivating us to improve the accuracy of Rewire's Full Vector mode.

Figure 7 shows box-plots of the designers' error score, measured by the average pixel distance. This shows that Ideal Vector has the lowest inter-quartile range, followed by Full Vector. We found no significant differences between any of the pairs of conditions demonstrating that Rewire's design assistance modes helped designers complete the tasks faster with no trade-offs in accuracy.

RQ2: Designers Ranking and Feedback on Modes

Figure 8 shows the designer's rankings of the design assistance modes from 1 (Most Preferred) to 4 (Least Preferred), showing that the Ideal Vector is the most preferred mode by 10 designers, followed by Smart-Snap. Smart-Snap was most consistently ranked second in the order, followed by Rewire's Full Vector mode. The factors that most affected designers' rankings were *perceived effort and time*, which were mentioned by 10 designers. However, the Full Vector mode required more fixes by designers so they perceived it as less accurate and more work than the Smart-Snap mode.

Designers' favorite part of having the ideal vector template was that it was more accurate, and required less effort. Designers liked not having to redraw shapes or manually align them (9 designers). P10 mentioned "*It was way easier! Now I can spend my time working on actual design.*". However, designers did not always think that they would want it in every scenario. P13 said "*It would be easier if I wanted to copy the exact screenshot. I usually change up colours, shapes, etc., so this wouldn't be helpful in that case.*"

The second most preferred mode was Rewire's *Smart-Snap*. The designers' favorite part *Smart-Snap* was that it made it easier and quicker to achieve a more accurate alignment (8 designers). P8 said *"I was able to get an idea of where exactly each element is properly placed with close to pixel perfect alignment.*" P5 said "The snapping guidelines are helpful and make for most accurate tracing of shapes - much better than doing them by hand." Five designers also mentioned Smart-Snap's help in drawing and matching the correct size for rectangles and other shapes. P11 said "*I really like the snapping guidelines because it takes the guesswork out of shape sizes and where items are on the page.*"

Smart-Snap seemed especially helpful for drawing rectangles and placing icon shapes, however, was less helpful for text. Snapping guidelines did not always appear in the correct place to help align new text boxes to the baselines of image text, so designers still had to align them manually. The snapping condition also did not help with matching any text or shape properties, so designers still found that part of the task to be challenging. Additionally, four designers mentioned that adding support for snapping to help obtain corner radii for rectangles would be useful. Currently, Smart-Snap only displays snapping guidelines for vertical and horizontal axes.

Rewire's *Full Vector* mode was most frequently ranked third. Designers mostly would not want to use it if the Ideal Vector

mode was available, since there were shapes and text that required fixes in the vectorized output. However, designers did like Full Vector mode's auto-generated shapes and text (9 designers), and felt that it required less effort overall than the screenshot-only mode (4 designers). P5 mentioned "*It was a nice balance of providing elements and still allowing the user to make decisions.*"

The most common dislike (6 designers) for *Full Vector* was having to manually fix issues in the auto-generated output. Rewire's Full Vector mode was not perfect. It may have required more cognitive load to detect and find the issues. P10 said "*It requires more brain computing to determine how much more needs to be done. I would prefer to have it draw only the objects it is most confident about.*" P1 said "*The fact that .. objects were not created accurately requires me to go through the auto-generated objects to make sure they are up to spec.*" Despite some designers' dislike of the fixes required in this mode, three designers mentioned it was easier to make these fixes than recreating from scratch.

Text shapes confused some designers because Rewire generates masked vector shapes instead of editable text boxes when it cannot detect text. We instructed them of this behavior in the warm-up, but several designers found this behavior confusing, and tried to edit the text before realizing Rewire had not generated the text box. Also, four designers mentioned it was difficult to distinguish Rewire's vector shapes from newly drawn shapes, despite the Rewire Highlights panel. Rewire's vector shapes are currently indistinguishable from hand-drawn shapes in the XD canvas. Thus it is difficult for designers to check them for accuracy. One designer suggested adding an indicator to distinguish them in the layers panel.

As Figure 8 shows, the *Screenshot-Only* mode was most commonly ranked last. None of the designers preferred it most. The designers most common dislike was the *lack of precision and accuracy* (7 designers). Designers mentioned needing to use more strategies (e.g, zooming, eyeballing, using guides) to ensure accurate alignment of shapes to the image, and disliked the precision in their output. Seven designers felt the task was more difficult, more work, and tedious compared to other modes. However, designers liked having more control and freedom with this mode. Five designers mentioned having more trust in its accuracy since it is their current method.

After ranking the modes, we had the designers describe scenarios in which they might find each mode most useful. For Smart-Snap, the designers thought it would be most useful for tracing and getting exact alignment (P12, P13), and when creating new shapes similar in size or layout to the screenshot but with different designs (P5, P13). They also thought it would provide more control than the other modes. Designers thought the Ideal Vector and Full Vector modes would be most useful for situations where the original assets were lost (P9, P13), having to match a UI or existing design language (P2, P3, P8), or in making quick mockups based an existing interface. However, accuracy was also important (P7, P11, P14). Designers felt that they would need to build trust in the auto-vectorized mode before integrating it into their design process. Screenshot-Only was only mentioned as useful when creating quick mockups if the designers did not care about accuracy or wanted a more loose recreation (P2, P8).

DISCUSSION AND FUTURE WORK

Rewire presents new tools for creating user interface designs based on example images. We see this work as an initial exploration of intelligent design assistance, and we see opportunities to develop more sophisticated tools in the future. One area we could explore is additional forms of design assistance. While evaluating Rewire, we discovered that interface design documents frequently contain complex hierarchies and shapes with ambiguous representations. Our system could infer multiple hierarchies and shape types and allow the designer to select from these candidate representations. Also, because achieving a perfect vectorization is difficult, we could build a user-in-the-loop system where designers can repair the vectorized output while training our system to improve its accuracy.

Additionally, we may be able improve the accuracy of our vectorization pipeline by applying deep learning techniques. Training an end-to-end network would likely require either a lot of training data (e.g., collected from online galleries and cleaned), or using data augmentation. Another option is to fine tune a pre-trained segmentation network (e.g., [2]) to handle interface screenshots. We would also like to explore better detection of natural images from interface shapes. To do this, we could potentially train a network to distinguish these segments. Also, small interface shapes elements frequently get filtered by our segmentation algorithms. We plan to explore methods of enhancing these low level techniques in the future. We also plan to explore the extension of Prefab's models [9] to discover more properties of shapes like shadows and gradients.

Finally, we acknowledge that tools like Rewire may unintentionally facilitate unsanctioned copying. Our formative work suggests that the tasks Rewire supports (creating derived designs, recreating vector designs when original assets are lost) are common practice in the design community and not viewed as "stealing". However, researchers and practitioners should consider the ethical implications of tools like Rewire when adopting them into their practices.

CONCLUSION

In this paper, we presented Rewire, a system that automatically infers a semantic vector-based representation of interface shapes from a pixel-based input screenshot. Rewire provides new forms of design assistance to ease the adaptation of example screenshots directly in designs. If designers can save time from recreating interface elements, they would potentially have more time to consider alternative designs, which would lead them to better final products [11]. We believe that systems like Rewire can enable us to explore new forms of intelligent design assistance enabling new possibilities in user interface design.

ACKNOWLEDGEMENTS

We would like to thank James Fogarty for his advice and feedback. This material is based upon work supported in part by the National Science Foundation under awards IIS-1314399 and IIS-1702751, and the National Science Foundation Graduate Research Fellowship under award DGE-1256082.

REFERENCES

1. Pablo Arbelaez. 2006. Boundary Extraction in Natural Images Using Ultrametric Contour Maps. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPRW '06)*. IEEE, 182–182. DOI: http://dx.doi.org/10.1109/CVPRW.2006.48
2. Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 12 (Dec 2017), 2481–2495. DOI: http://dx.doi.org/10.1109/TPAMI.2016.2644615
3. Dana H Ballard. 1981. Generalizing the Hough transform to detect arbitrary shapes. *Pattern recognition* 13, 2 (1981), 111–122.
4. Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *CoRR* abs/1705.07962 (2017). http://arxiv.org/abs/1705.07962
5. William Buxton. 2007. *Sketching User Experiences: getting the design right and the right design*. Elsevier/Morgan Kaufmann, San Francisco, CA; Amsterdam.
6. Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.
7. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 767–776. DOI: http://dx.doi.org/10.1145/2984511.2984581
8. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1525–1534. DOI: http://dx.doi.org/10.1145/1753326.1753554
9. Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 969–978. DOI: http://dx.doi.org/10.1145/1978942.1979086
10. Morgan E. Dixon, Gierad Laput, and James A. Fogarty. 2014. Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, New York, USA, 2231–2240. DOI: http://dx.doi.org/10.1145/2556288.2556979
11. Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. 2010. Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-efficacy. *ACM Transactions on Computer-Human Interaction (TOCHI)* 17, 4, Article 18 (Dec. 2010), 24 pages. DOI: http://dx.doi.org/10.1145/1879831.1879836
12. Álvaro González, Luis M. Bergasa, J. Javier Yebes, and Sebastián Bronte. 2012. Text Location in Complex Images. In *Proceedings of the International Conference on Pattern Recognition (ICPR '12)*. IEEE, 617–620.
13. Scarlett R. Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P. Bailey. 2009. 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 (CHI '09)*. ACM, New York, NY, USA, 87–96. DOI: http://dx.doi.org/10.1145/1518701.1518717
14. Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092. DOI: http://dx.doi.org/10.1145/2470654.2466420
15. Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 2197–2206. DOI: http://dx.doi.org/10.1145/1978942.1979262
16. Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R. Klemmer. 2010. Designing with Interactive Example Galleries. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2257–2266. DOI: http://dx.doi.org/10.1145/1753326.1753667
17. Y. Li and H. Lu. 2012. Scene Text Detection via Stroke Width. In *Proceedings of the International Conference on Pattern Recognition (ICPR '12)*. IEEE, 681–684.
18. Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE, 248–259. DOI:http://dx.doi.org/10.1109/ASE.2015.32
19. Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2014. Learning Layouts for Single-Page Graphic Designs. *IEEE Transactions on Visualization and Computer Graphics* 20, 8 (Aug 2014), 1200–1213. DOI:http://dx.doi.org/10.1109/TVCG.2014.48
20. Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2015. DesignScape: Design with Interactive Layout Suggestions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1221–1224. DOI: http://dx.doi.org/10.1145/2702123.2702149

21. Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. 2005. The CIEDE2000 Color-Difference Formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research and Application* 30, 1 (2005), 21–30. DOI: http://dx.doi.org/10.1002/col.20070

22. Ray Smith. 2007. An Overview of the Tesseract OCR Engine. In *International Conference on Document Analysis and Recognition (ICDAR '07)*, Vol. 2. IEEE, 629–633. DOI: http://dx.doi.org/10.1109/ICDAR.2007.4376991

23. Holm Sture. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70. http://www.jstor.org/stable/4615733

24. Pengfei Xu, Hongbo Fu, Takeo Igarashi, and Chiew-Lan Tai. 2014. Global Beautification of Layouts with Interactive Ambiguity Resolution. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 243–252. DOI: http://dx.doi.org/10.1145/2642918.2647398

25. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 183–192. DOI: http://dx.doi.org/10.1145/1622176.1622213