Non-Photorealistic Animation and Rendering
Pierre Bénard and Holger Winnemoeller (Editors)

EXPRESSIVE 2016

Data-Driven Iconification

Yiming Liu†[1] Aseem Agarwala†[1] Jingwan Lu[2] Szymon Rusinkiewicz[3]

[1]Google Inc. [2]Adobe Research [3]Princeton University

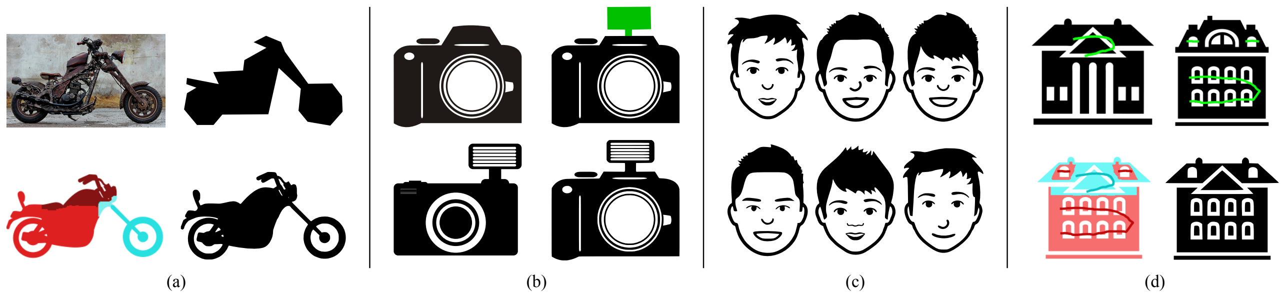

Figure 1: Four icon customization workflows supported by our algorithms. (a) ***Sketch-based pictogram modeling:*** *given an input photo (top left) of a motorbike, the user sketches a polygon (top right) over the photo; this sketch, along with the keyword "motorbike," are the only user inputs (the photo is* not *used). Our method remixes partially similar pictograms (bottom left) to create a pictogram (bottom right) that matches the user's sketch. (b)* ***Sketch-based pictogram editing:*** *starting with an existing pictogram of a camera (top left), the user sketches a flash (top right, green blob) on top of the camera, and our method remixes the partially similar pictogram (bottom left) to create a pictogram of camera equipped with flash (bottom right). (c)* ***Pictogram hybrids:*** *Starting with several stock pictograms of boy faces (shown in Fig. 13a), our method creates random yet visually appealing hybrids. (d)* ***Pictogram montage:*** *guided by the user's scribbles (top, green), our method helps the user merge two pictograms while retaining the user-selected parts (bottom).*

Abstract

Pictograms (icons) are ubiquitous in visual communication, but creating the best icon is not easy: users may wish to see a variety of possibilities before settling on a final form, and they might lack the ability to draw attractive and effective pictograms by themselves. We describe a system that synthesizes novel pictograms by remixing portions of icons retrieved from a large online repository. Depending on the user's needs, the synthesis can be controlled by a number of interfaces ranging from sketch-based modeling and editing to fully-automatic hybrid generation and scribble-guided montage. Our system combines icon-specific algorithms for salient-region detection, shape matching, and multi-label graph-cut stitching to produce results in styles ranging from line drawings to solid shapes with interior structure.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation

1. Introduction

Pictograms, or *icons*, are abstracted pictorial representations of objects or concepts. Good pictograms offer an efficient, unambiguous, instantly recognizable visual language of concepts, from "global warming" to "cat," that crosses language boundaries. They are commonly used in graphic designs, infographics, explainer videos, logos, and other forms of visual communication. Their ubiquity has given rise to collections such as The Noun Project `<http://thenounproject.com>`, a database of nearly 100,000 curated pictograms contributed by a wide array of designers. The popularity of The Noun Project demonstrates the growing market for customized variations of icons. Novice users are willing to buy a specific face icon resembling their own facial features to be used on social networks. Professional Artists might find a variety of icons of the same concept useful for their design ideation process.

† Most of this work was done when Yiming Liu was a PhD student at Princeton University and an intern at Adobe Research, and Aseem Agarwala was with Adobe Research.

While these hand-made icons are both beautiful and useful, they are a small subset of the much larger space of plausible icons. A long-standing challenge in computer graphics is to synthesize large varieties of plausible shapes from a few hand-created examples [KCKK12, RHDG10]. For example, the existing "horse" icons in the dataset sample the larger space of all possible "horse" icons; is it possible to generate even more samples through interpolation of the existing ones? Can a user control this interpolation to create customized icons with particular properties?

This paper describes a system for user-driven synthesis of customized icons. Our approach includes a number of ways to generate variations for a particular class of icons (e.g., "horse"), which all start by downloading a few exemplars fetched by keyword. *Pictogram Hybrids* (Section 4.3 and Figure 1c) generate icon variations completely automatically by randomly combining parts of the exemplars. The rest of our methods allow for user control. *Sketch-Based Pictogram Modeling* (Section 4.1 and Figure 1a) takes a rough sketch of an icon from the user (which may be traced over a photo), and assembles parts of exemplars to match the sketch. *Sketch-Based Pictogram Editing* (Section 4.2 and Figure 1b) allows a user to select a particular icon and then remove parts and/or add sketched regions; the algorithm finds parts of other exemplars to satisfy the user edits. Finally, *Pictogram Montage* (Section 4.4 and Figure 1d) allows a user to select multiple exemplars and combine parts of them into a new, plausible pictogram.

Our system allows multiple workflows because users have different reasons for customizing icons. Some users may simply want to be inspired by automatically generated variants; others might have a rough shape from a photograph they wish to mimic; and still others may like a particular icon but wish to change a local detail. All of these workflows are supported by our framework. We describe the rest of our algorithms in the context of our most challenging workflow, Sketch-Based Pictogram Modeling, and then show how the different workflows can be reduced to this variant.

We must solve two technical problems to accomplish the goal of customizing icons: search and remixing. First, we find parts of exemplar icons from a database (Section 3.1) that match salient parts of the user sketch, allowing for some geometric transformation. Though similar to the problem of partial shape matching [VH01], we have the added complication that the user-drawn sketch will generally be rougher and less detailed than the desired pictogram. We therefore decompose the sketch into a number of salient regions (Section 3.2), and then use approximate sliding-window matching (Section 3.3) to find candidate exemplar parts.

We then combine the candidate parts (Section 3.4) into a plausible, seamless whole that matches the user-drawn shape; for this step we use a Markov Random Field (MRF) optimization. We describe how to limit the space of matching pictograms to generate results in a variety of coherent styles (Section 3.5). Our results (Section 4.1) demonstrate many cases in which this algorithm is able to beautify user-provided pictograms while introducing detail, including interior detail that was not present at all in the input.

2. Related Work

There are a number of techniques for helping users create 2D sketches. One approach is to provide a content-sensitive visual reference [LZC11, JAWH14]; in this case, the user-drawn sketch is the final artifact. Another approach is to automatically beautify the sketch [Zit13, BTS05]. Our approach is instead to replace a user-drawn shape with a combination of parts from professionally drawn art.

Combining parts of existing artwork has been explored in a number of domains. For example, combining parts of photographs [ADA*04, HE07, CCT*09] and manipulating photos [GCZ*12] are common operations, though the domain of pictograms requires very different design choices than natural images. First, the lack of color and texture requires us to focus on shape alone. Imperfections in how contours match up cannot be masked by texture, and even modest deformations to the shape can ruin the beautiful forms designed by artists. Therefore, we must devote significant effort at both the retrieval and graph-cut stages to ensuring that the parts we remix match up seamlessly. Second, because our source icons are designed by artists, and not by the public at large, there will be a significantly smaller number of them available for any query: dozens or hundreds, as opposed to millions of amateur photographs. Therefore, we often cannot find an almost-perfect match for a query, and must focus on remixing exemplars with large variation, rather than on warping already-good retrieval results to match a query's details.

Combining parts of retrieved results has also been explored in data-driven 3D modeling systems [FKS*04]; however, significant user effort is required. More recent systems [CKGK11] reduce user effort, but require a pre-segmented and labeled 3D model database. Such model annotation would be difficult for pictograms, which are more abstract than 3D shapes. Most similar to our problem are photo-guided techniques that help users build a 3D model similar to a reference photograph using parts from a model database [XZZ*11, SFCH12]; again, these require aligned and segmented 3D models.

Another approach to remixing parts of objects into new ones is to generate large numbers of combinations without any user guidance. This idea has been explored for 3D models [OLGM11, KCKK12, JTRS12], and 2D imageries [RHDG10, HL12]. These 2D methods can produce pictograms, but do not allow user control.

Part of our technical solution involves partial 2D shape matching. In computer vision literature [VH01], shape matching is used to find objects in photographs via their silhouettes. Shapes are often matched partially in order to handle occlusions or deformation, but even partial shape matching is typically evaluated for whole-object detection tasks [ML11]. In contrast, we are truly interested in finding partial shapes. Shape matching in our case has two other significant differences from that in object recognition. First, icons are typically in canonical positions, so we do not need to allow for arbitrary rotations (we do allow for reflections, scale and translation). Without rotation, easier matching methods can be used, such as sliding windows. Second, the user-drawn sketches typically contain less detail than the database icons; we therefore first find the most salient regions of the sketch, and focus the matching on those.

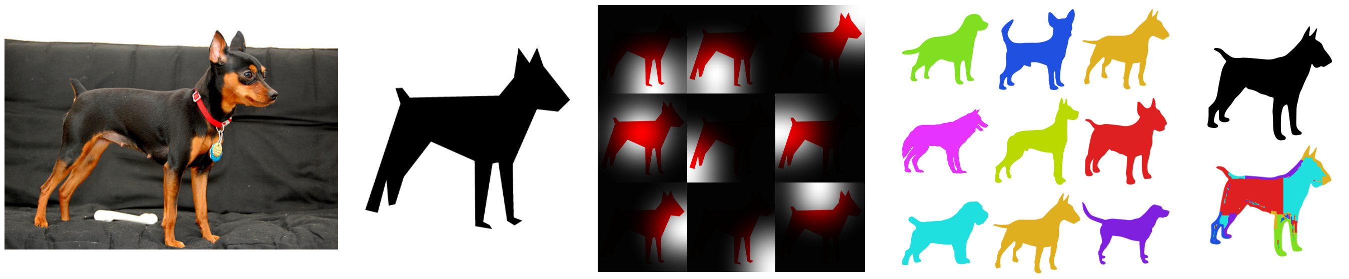

Figure 2: *An overview of our sketch-based pictogram modeling algorithm for a* dog *query. The user first selects a photo (a) illustrating the desired concept, and sketches a rough polygon (b) over it. Our algorithm detects salient regions (c) in the polygon, then uses sliding-window matching to find the most similar exemplar pictograms (d) for each region (only one for each region shown here). These are remixed and blended to create a final pictogram (e, first row) that matches the user's intent. We color-code each pixel to indicate which icon the pixel comes from (e, second row).*

Once shape regions are matched, we use a graph-cut-based algorithm to seamlessly combine the pictograms, similar to algorithms used in image compositing [BVZ01, KSE*03, ADA*04]. However, our energy functions are customized to the pictogram context. Given the user-drawn outer contour, we use signed distance functions to propagate shape information globally. We also use larger windows (7×7) for the pairwise smoothness cost in graph cuts, since higher-order smoothness artifacts are more noticeable for vector art.

3. Algorithm

We first describe our full pipeline for the most challenging workflow, Sketch-Based Pictogram Modeling. We then show in Section 4 how the other workflows can be accomplished with the same system with minor technical modifications.

Fig. 2 illustrates the modeling workflow. We begin by asking the user to sketch a solid polygon P (Fig. 2b), which might be obtained by drawing the object of interest over a photo (Fig. 2a), or by drawing free-hand. For the user's convenience, we expect P to describe only the rough outline of the desired pictogram — our algorithm will add details to the contour and fill in the interior structure. We then ask the user for a keyword K (*dog*, in this case) to describe the object class, and use K to retrieve a collection of at most 200 pictograms from The Noun Project (Sec. 3.1) using their API without manual curation.

Since the user's sketch is coarse compared to the desired result, we first detect the detailed, salient regions in P that should control the matching process (Sec. 3.2). We then find the best matches $\{I_i\}$ for each region (Sec. 3.3). These matches are combined using multi-label graph-cuts (Sec. 3.4) to produce a new pictogram (Fig. 2d) that mimics P. We restrict matches to come from exemplars of similar style (Sec. 3.5) in order to ensure consistency in our output.

Our remixing algorithm operates in the pixel domain, then re-vectorizes the final result. This might seem wasteful, since the input SVG pictograms are vector, not raster, drawings. However, we chose pixel-domain processing for two reasons. First, it is difficult to remix interior structures consistently with the contours in the vector domain. Second, the deformations required to merge contours in the vector domain often lead to unnatural appearance or even break semantics. We note that recent work on stroke stylization [LBW*14] also made the same choice of raster-domain processing followed by vectorization, for the same reasons.

3.1. Fetching Data

Our system obtains source pictograms for remixing from The Noun Project, a large user-contributed but curated online pictogram repository. In order to restrict our remixing to models with detail appropriate for the given class, we query the repository with the user-provided keyword, and fetch pictograms in SVG format via its API. We only focus on the pictograms consisting of black and white solid shapes and lines, which covers the majority of the repository.

The SVG file of a pictogram fetched from The Noun Project represents a tree, with geometric primitives stored at its leaves and interior structure representing grouping information. However, the structure of the tree proves difficult to exploit for semantic grouping, for a few reasons. First, most artists do not segment the pictogram into semantically-independent parts. In other words, the tree structure in the SVG file is neither consistent nor reliable. Second, pictograms are often 2D projections of 3D objects; different pictograms, even with the same keyword, may represent projections from different views and with different occlusions. This makes it difficult to extract a *semantic* tree model to represent the hierarchical structure of parts in a pictogram, and hence to infer the semantic correspondence between pictograms. Therefore, we are only able to consider a pictogram as an unstructured collection of shapes.

3.2. Salient Region Detection

Given a sketch polygon P drawn by the user, we first find a number of overlapping salient regions. Intuitively, we assume that each noticeably convex or concave local structure on the polygon boundary reflects the user's intent — otherwise, the user would not have included that detail. We detect these local structures by first computing a salience map, and then selecting local regions from this map. The salience map is a sum of two terms that measure *turning angles* and *black/white balance*, respectively.

The turning-angle term measures geometric salience by aggregating the turning angles of polygon edges in local regions. In practice, we first rasterize P onto an $m \times m$ bitmap I_P, where $I_P(x,y) = 0$ (black) for all pixels inside of P, and $I_P(x,y) = 1$ otherwise. In our implementation, we set $m = 256$. We compute the (unsigned) exterior angle $\theta(x,y)$ for each pixel at a polygon vertex, and let $\theta(x,y) = 0$ for other pixels. Then, the aggregated exterior angle is the convolution of θ and a smoothing kernel M:

$$T = \frac{\theta * M}{\max(\theta * M)}, \tag{1}$$

normalized to ensure that $\max(T) = 1$. The mask M represents the extent of a region (e.g., the nine patterns shown in Figure 2c); the specific design of M is discussed below.

A good salient region should be neither mostly inside P nor mostly outside P, to prevent trivial matching results (all black/white pixels). The second term in the salience map therefore measures black/white balance — i.e., the balance between pixels inside and outside of P:

$$B = \left|(I_P - 0.5) * M\right|, \tag{2}$$

where both the subtract and absolute operations are element-wise. Thus, B will be near 0 when about half the pixels in the neighborhood are black.

We compute the final salience for each pixel by combining the turning-angle score and the black/white balance score:

$$S(x,y) = T(x,y) - \lambda_B B(x,y), \tag{3}$$

where $\lambda_B = 0.1$. This weighs $T(x,y)$ more than $B(x,y)$, because our ultimate goal is to detect geometric salience.

The design of our mask M has several motivations. First, to prevent false-positive matches due to overfitting local structures, M should be a rather large mask. In addition, we would like regions to have significant overlap with adjacent ones, so that selected pictograms will have similar geometry in overlapping regions. Finally, since a portion of boundary area will be cut in the remixing step, M should focus mostly on matching structures in its center. Therefore, M should be constant in the middle and fall off smoothly. In particular, we take M to be a 100×100 square smoothed by a Gaussian with $\sigma = 40$.

For the number of salient regions, we choose 9; we find that this strikes a balance between matching details in P while still maintaining much of the global structure of the exemplars. In addition, this covers the whole image with considerable overlap between regions. These 9 regions should have the highest scores, but be located at least $\Delta_S = 75$ pixels apart. We find these regions in a greedy fashion: in each round, we pick a region centered at a valid pixel with the highest score, then label pixels within Δ_S as invalid. With this strategy, the first regions we obtain often contain one or more of the most-salient local structures, and also have the best balance between pixels inside and outside the polygon. These regions help us to capture the important large-scale structure of the user's intent. The last regions we obtain usually contain just a single small salient local structure, and are often less balanced between inside/outside pixels. This is helpful to capture local details.

For each salient region, we create a mask M_i $(1 \leq i \leq 9)$ with $m \times m$ pixels to be used for sliding-window matching. Specifically, for each pixel (x,y) inside the region, we let $M_i(x,y)$ equal the corresponding value of M; for other pixels, we let $M_i(x,y) = 0$. Fig. 2c shows the masks we generate for the 9 salient regions.

3.3. Sliding-Window Matching

For each salient region, we now wish to find the exemplar pictograms that match the query most closely in that region (i.e., as weighted by the mask M_i). To formulate the matching problem, we must define both the matching function (i.e., when two pictograms are considered similar) and the space of transformations over which to search.

The most natural choice for a matching function is the difference between (black/white) rasterized images, weighted by M_i. However, it introduces large penalties for even slight misalignment, reducing our ability to match roughly-drawn user sketches to detailed icons from the database. Instead, we would like to allow small mismatches between the sketch and a database icon, while still penalizing large misalignment. We therefore choose to match pictograms by comparing *truncated signed distance fields* (TSDF), which have been used in previous texture synthesis approaches [LH06]. For each pixel of both the sketch P and each retrieved pictogram E_j, we find its signed distance (positive outside, negative inside) to the nearest point on the exterior contour. To avoid over-penalizing mismatches, we clamp the signed distance value to some range $[-d_{max}, d_{max}]$; we experimentally found that $d_{max} = 6$ worked well. Finally, we normalize the TSDF by dividing by m. We formulate the matching function as the weighted squared distance between a normalized TSDF image D_P for the query and D_j for a pictogram E_j:

$$\delta(\mathcal{T}) = \sum_{x,y} M_i(x,y) \left[\mathcal{T}\big(D_j(x,y)\big) - D_P(x,y)\right]^2, \tag{4}$$

where $\mathcal{T}$ is an appropriate transformation.

We next need to choose a class of transformations over which to search. Previous shape-matching systems are complicated by the need to consider the full set of rigid-body transformations; rotations in particular lead to more complex shape matching. In our application, however, we observe that most pictograms have a clearly defined upright orientation, which causes rotation to break semantics. Therefore, we restrict our transformations $\mathcal{T}$ to be translations, which greatly simplifies the matching problem. In fact, the sliding-window computation can be effectively accelerated via the Fast Fourier Transform (see Appendix A).

In practice, we allow two additional classes of transformations to improve matching. First, we allow *horizontal* (but not vertical) flipping. Second, we allow modest *uniform* (but not anisotropic) scaling, restricted to factors of 0.9 and 1.1. Artists tend to draw fewer details on smaller parts of objects. Very aggressive scaling potentially leads to results that combine parts with different levels of details, which is what we want to avoid. The search over these transformations is implemented by creating additional flipped and/or scaled versions of each retrieved pictogram. Notice that, we do not warp or deform the retrieved icon to better match the query polygon, since spatially varying deformation often makes the results worse. In contrast to photos, monochrome icons have only shapes and no texture or color, making even a small, incorrect shape deformation visually stand out.

Thus, for each salient region we obtain a ranked list of exemplars, together with translation/flip/scale, that best match that region. We keep the 3 best matches for each region. Fig. 2d shows one of them per region. The resulting $9 \times 3 = 27$ transformed exemplars serve as sources for our remixing stage.

3.4. Pictogram Remixing

The remixing stage needs to produce a plausible output pictogram respecting the user sketch. Specifically, the remixing should (1) yield a contour similar to the user's query P; (2) synthesize interior structure present in the exemplars (but not the query); and (3) produce a seamless result, both on the outside contour and on inside structure.

We formulate remixing as a multi-label graph-cut problem. Specifically, the result is modeled as a 4-connected 2D lattice containing $m \times m$ sites, one per pixel. Each site has an indicator label $L(i)$ that takes one of 27 values, corresponding to the exemplars. The energy function is modeled as the sum of a unary data cost and pairwise smoothness cost:

$$F(L) = \sum_{i \in V} F_D\big(i, L(i)\big) + \lambda_S \sum_{(i,j) \in E} F_S\big(i, j, L(i), L(j)\big), \tag{5}$$

where V is the set of sites on the lattice and E is the set of edges connecting adjacent sites on the lattice. In this formulation, the data cost F_D measures the similarity to P, and the smoothness cost F_S measures the smoothness of remixing.

Data Cost. We expect the contour of our remixing result to be similar to P. Since P only describes the rough shape of the user's desired result, we tolerate small offsets between the result contour and P. For this reason, we use the truncated signed distance field of the contour to measure data cost. Specifically, we compute the TSDF for P and *only the external contour* (i.e., ignoring internal structure) of $\{I_k\}$ to obtain D_P and $\{D_k\}$. At each site i, the data cost is then defined as the difference between P and the exemplar:

$$F_D\big(i, L(i)\big) = \big|D_P(i) - D_{L(i)}(i)\big|. \tag{6}$$

Smoothness Cost. Since we expect the remixing result to be seamless, we would like to avoid visible offsets when remixing pictograms. For this reason, we use the image pixel color instead of the signed distance field in the smoothness cost. Specifically, we collect a 7×7 neighborhood $N_k(i)$ for each pixel in each exemplar I_k. The smoothness cost is defined as

$$F_S\big(i, j, L(i), L(j)\big) = \begin{cases} 0 & \text{if } L(i) = L(j) \\ \max(\varepsilon, f_n + f_{int}) & \text{otherwise,} \end{cases} \tag{7}$$

where f_n is the distance between neighborhoods:

$$f_n = \big\|N_{L(i)}(i) - N_{L(j)}(i)\big\|_2 + \big\|N_{L(i)}(j) - N_{L(j)}(j)\big\|_2, \tag{8}$$

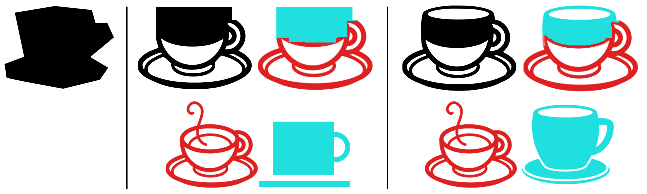

Figure 3: *Patch-based vs. pixel-based smoothness cost. The user sketches a cup (a, top). We compare graphcut results with pixel-based (b) and patch-based (c) smoothness cost. Patch-based smoothness results in a more natural connection of the curves on the two exemplars. Note that the weight placed on smoothness, λ_S, is adjusted to 10 in (b) and 1 in (c) so that both results use exactly two exemplars.*

and f_{int} is a penalty for cutting through interior structure. Specifically, we set $f_{int} = 0.5$ if either $I_{L(i)(i)}$ or $I_{L(j)}(j)$ is interior structure (a white pixel inside the contour), and set $f_{int} = 0$ otherwise. Finally, if two different labels are remixed, even if the remixing is perfectly seamless and avoids cutting through the interior structure, we still consider it worse than not remixing. Therefore, we clip the lower bound of F_S to $\varepsilon = 0.01$ when $L(i) \neq L(j)$. Notice that, though not explicitly handled, plausible interior structure are synthesized. The use of 7×7 neighborhood prevents the merging of inconsistent interior structures during graph-cut. For example, pixels on the boundaries of square windows and circular windows (Fig 15) have different neighborhoods. Attempting to merge them yield high smoothness cost and is therefore avoided.

Symmetry Constraint. For pictograms exhibiting reflectional symmetry, *e.g.* a lamp, face, or airplane, we would like to ensure that each pixel is taken from the same exemplar as its symmetric counterpart. To this end, we connect each site i to r_i, where i and r_i are symmetric with respect to a symmetry axis. The smoothness cost on this edge is defined as:

$$F_S\big(i, r_i, L(i), L(r_i)\big) = \begin{cases} 0 & \text{if } L(i) = L(r_i) \\ +\infty & \text{otherwise,} \end{cases} \tag{9}$$

In our experiments, we find it sufficient to restrict the symmetry axis to horizontal, vertical, or diagonal lines through the center. Since each pictogram is rescaled to its tight bounding box and centered with respect to the canvas, as long as its outer boundary is symmetric (which is true for most cases), its true symmetry axis will pass through the center. (The same assumption is also made by previous work such as Structured Image Hybrids [RHDG10].)

It is not difficult to prove that our smoothness cost is a metric in the space of labels. That enables us to use the α-expansion algorithm [BVZ01] to approximately minimize $F(L)$.

Impact of λ_S. The parameter λ_S controls the balance between data and smoothness cost. Intuitively, a large value of λ_S encourages smoothness of remixing, at the expense of similarity to the user's sketch polygon. A small value of λ_S makes the remixing result more similar to the user's sketch polygon, but artifacts tend to emerge. In practice, we observe that when $\lambda_S \geq 3$, the remixing result is most likely a single pictogram that is most similar to the

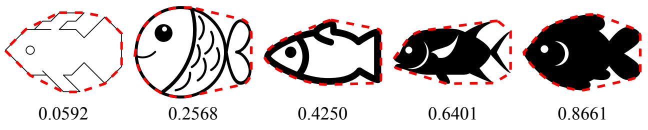

Figure 4: *Five* fish *pictograms enclosed by their convex hulls (red dashed polylines), and the resulting blackness values.*

user's sketch polygon. When $\lambda_S \leq 0.01$, the remixing result has contributions from almost all exemplars, but has many artifacts. We observe that there are often multiple values of λ_S that yield plausible but different results, and the ideal value depends on how closely the user wishes to match the input sketch. In practice, we sample 8 values in the range $[0.03, 0.75]$, and run the graph-cut algorithm with each sampled value. All of these are presented to the user.

The Benefit of Patch-Based Smoothness Cost. A key difference between our remixing algorithm and Photomontage [ADA*04] is that we use 7×7 patches instead of pixels in our smoothness cost. Given the fact that the user's sketch is usually rough, this change is critical to avoid high-order artifacts, e.g. in the slope and curvature of strokes and contours. Fig. 3 demonstrates an example in which the user sketches a cup. Although we expect the remixing results to differ from the user's sketch, they should still have a natural appearance. However, we observe that the pixel-based smoothness cost results in an unnatural connection of a straight line and a curve on the edge of the cup. In contrast, our patch-based smoothness cost yields a natural remixing.

The neighborhood size in the smoothness term is a compromise among quality, speed, and memory consumption. Generally, a larger neighborhood is able to keep higher-order structures, but it makes graph-cuts slower and less memory efficient. In our experiments, we find that 7×7 neighborhoods work well enough in most cases, while also having acceptable speed.

Post-processing. As noted earlier, all of our matching and remixing operations are performed on rasterized versions of the pictograms. We use the open-source `potrace` package `<http://potrace.sourceforge.net>` to vectorize the result.

3.5. Style Consistency

Pictograms from The Noun Project come in various styles. Some are solid shapes, some have rich interior structure, and some are line drawings. We design our algorithm to only remix pictograms of similar styles, for two reasons. First, mixing multiple styles usually produces implausible results. Second, restricting the algorithm to one style at a time allows us to produce multiple results in various styles, offering the user a greater selection of results to match his or her intent.

There is an existing method to measure similarity of illustration style [GAGH14]; however, it is overkill for the limited range of "pictogram" styles. We instead use a simple *blackness* feature to measure the style of a pictogram. We define blackness as the proportion of black pixels in the convex hull of the pictogram. Intuitively, this helps us distinguish among solid pictograms, those with rich interior structure, and line drawings. In addition, we observe that among line-drawing pictograms with the same keyword, blackness has a strong correlation with stroke width. In general, if we sort pictograms with the same keyword by their blackness values, the first ones are line drawings with very thin strokes. Strokes become thicker as blackness increases. Then pictograms with rich interior structure appear. Purely solid pictograms come last. Some examples are shown in Fig. 4.

Blackness values lie between 0 and 1, and we create four overlapping intervals to cover this range, as shown at right. We associate a "style" with each interval and generate separate results for each one by restricting matching to pictograms with blackness falling in that interval. In general, each style for which there are sufficient pictograms in the database yields a plausible result.

style	blackness
lightest	[0.0 .. 0.4]
light	[0.2 .. 0.6]
medium	[0.4 .. 0.8]
dark	[0.6 .. 1.0]

4. Workflows

We first show results of the algorithm we just described for Sketch-Based Pictogram Modeling. Then, we describe minor modifications to this algorithm to support three other workflows, and provide results for them, as well.

4.1. Sketch-Based Pictogram Modeling

We examine a set of test cases to show how our algorithm works. For each test case, we run our graph-cuts algorithm with each of the four blackness intervals and eight values of λ_S from 0.03 to 0.75. Ideally we would produce four results, with four different styles, for each test case. However, there are often not enough pictograms for every style for each keyword, and our algorithm fails to produce good results for those styles. So, we only demonstrate styles that yield plausible results. Our results figures contain a mix of different λ_S and style parameter settings; we use a legend bar at the top of each figure to identify the parameters used. We refer readers to our supplementary materials to find remixing results with all four styles and all eight values of λ_S.

The figures show how each result is assembled by assigning a distinct hue to each source pictogram. Often, several exemplars come from the same retrieved pictogram, but with different offset, flip, or scale. In this case, we use the same hue with different lightness (in HSL color space). Note that unless λ_S is large, our results may contain many tiny parts from different exemplars on the interior, which reduces data cost for pixels inside the contour. While this has no impact on the appearance of the remixing result, it makes the visualization more difficult to interpret. So, for clarity, we only include exemplars that contribute significantly to the result.

Motorbike. Fig. 1a demonstrates a test case of motorbikes. Note how our algorithm uses additional pictograms to make the motorcycle "chopped".

Bicycle. In Fig. 5a, the user sketches a polygon for a bicycle with a very low seat and without top tubes, and would like to remix pictograms from the Noun Project to mimic this appearance. Fig. 5(b-d) demonstrates three pictograms in different styles generated by our algorithm. In Fig. 5b, the first exemplar already has a low seat, but it has a top tube; the second exemplar is the opposite

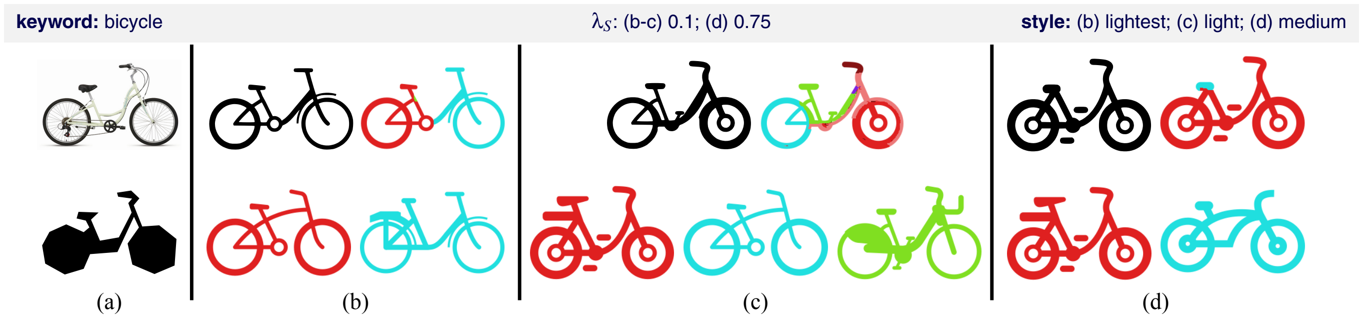
keyword: bicycle
λ_S: (b-c) 0.1; (d) 0.75
style: (b) lightest; (c) light; (d) medium
(a)
(b)
(c)
(d)

Figure 5: Bicycle. *(a) The user sketches a polygon to roughly represent the shape of a bicycle in a photo. (b-d) Three remixing results in different styles (i.e., blackness values). The first row contains the result, and a color visualization of how the result is assembled. The second row contains the exemplars that contribute a significant portion of the result.*

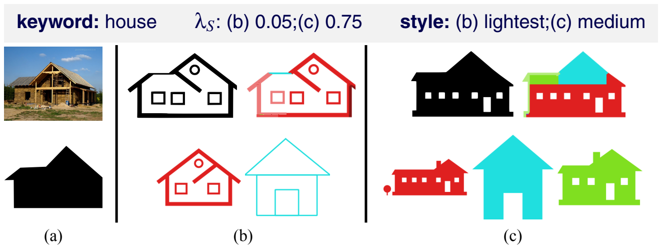
keyword: house
λ_S: (b) 0.05;(c) 0.75
style: (b) lightest;(c) medium
(a)
(b)
(c)

Figure 6: A House *in three different styles.*

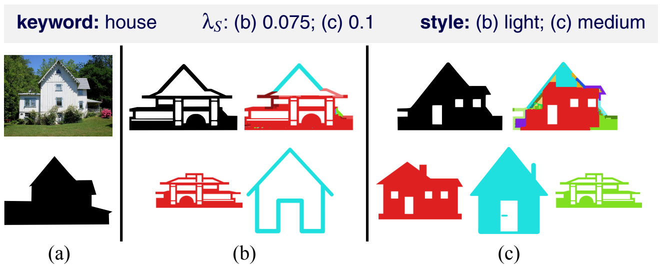

Figure 7: *Another* House *in two different styles.*

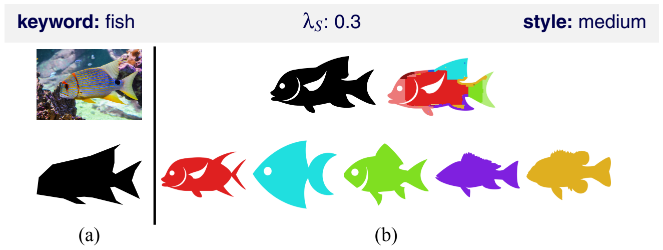

Figure 8: Fish. *Parts of many exemplars are combined to match the sizes of the fins and tail.*

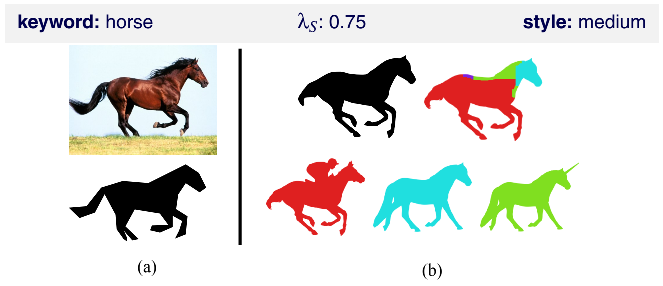

Figure 9: *A* Horse *pictogram.*

case. Our algorithm remixes them seamlessly, to produce a result that perfectly matches the user's intent. Fig. 5c collects parts from three pictograms, yet produces a seamless result with a low seat and no top tube. In Fig. 5d, our algorithm cuts a low seat from the second exemplar, and pastes it onto the first exemplar, to obtain a visually plausible result.

House. Fig. 6 demonstrates remixed house pictograms in three styles: line drawings, solid shapes with rich interior structure, and solid shapes. Note how our algorithm uses two identical line drawings of the house with offsets to mimic a wider house in Fig. 6(b). Fig. 7 demonstrates a different house in two styles: line drawings and solid shapes with rich interior structure.

Fish. In Fig. 8a, the user sketches a fish with a large dorsal fin on top, two smaller fins on the bottom, and a fat tail. With $\lambda_S = 3$, our algorithm would just return the red pictogram in Fig. 8b as the result, matching the positions but not the sizes of the fins. With a smaller value of $\lambda_S = 0.3$, our algorithm uses several additional pictograms to produce a result with correctly-sized fins and tail (see Fig. 8b). Also, note how plausible internal details are included that the user does not need to sketch on her own.

Horse. Fig. 9 demonstrates a test case of horses. Note how our algorithm uses additional pictograms to remove the rider from the back of the horse in Fig. 9.

4.2. Pictogram Editing

Our system also enables editing existing pictograms rather than sketching from scratch. Fig. 10 shows an example.

We enable editing by encouraging our remixing algorithm to choose the initial pictogram E_0 outside the changed area, i.e., the union of red and green parts, and penalizing E_0 inside that area. To this end, we create a mask D_E by computing a truncated distance field outside the union of the red and green parts (Fig. 10c). D_E is normalized by the truncation threshold $t = 25$ to ensure its maximum value equals 1. To focus our salient region search on the

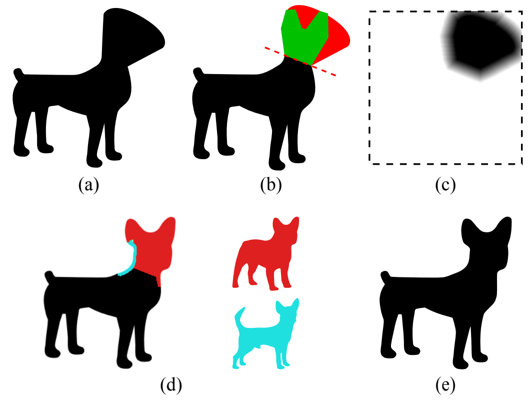

Figure 10: *Pictogram editing. Starting from an existing pictogram (a), the user removes the red region using a red, dashed line and sketches the new, desired shape with a green polygon (partially occluding the red region). Our system generates a mask (c) for the changed region, and exemplars are retrieved and remixed (d) based on this mask to generate an edited pictogram (e).*

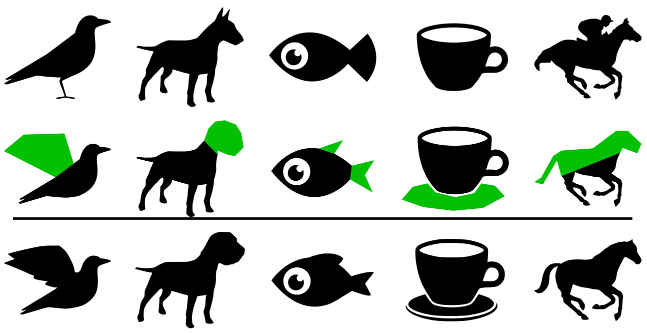

Figure 11: *Pictogram editing results. Row 1: initial pictograms; Row 2: the user's edits; Row 3: edited results.*

changed area, we add a term to S with a high weight $\lambda_D = 100$:

$$S'(x,y) = S(x,y) + \lambda_D \left[(1 - D_E) * M \right]. \tag{10}$$

We rule out pictogram E_0 when selecting exemplars during sliding-window matching, but add E_0 back in as label 0 during graph cuts. We then add a penalty P_E to the data cost:

$$P_E(i, L(i)) = \begin{cases} w_E \left[1 - D_E(i) \right] & \text{if } L(i) = 0 \\ w_E \, D_E(i) & \text{otherwise.} \end{cases} \tag{11}$$

For pixels inside the changed area, $P_E = w_E$ if $L(i) = 0$, so E_0 is penalized. The penalty is reduced as pixels get more distant from the changed area until the t threshold is crossed and the penalty becomes zero. Similarly, the other source pictograms are penalized outside the changed area, but are encouraged inside. Weight $w_E = 0.1$ controls the balance between this penalty and the original data term. The parameters w_E, t, and λ_D are set experimentally.

We show a number of results of editing existing pictograms to change one or more parts in Fig. 11. Notice how the algorithm uses parts of other icons to add detail to the user's rough sketches.

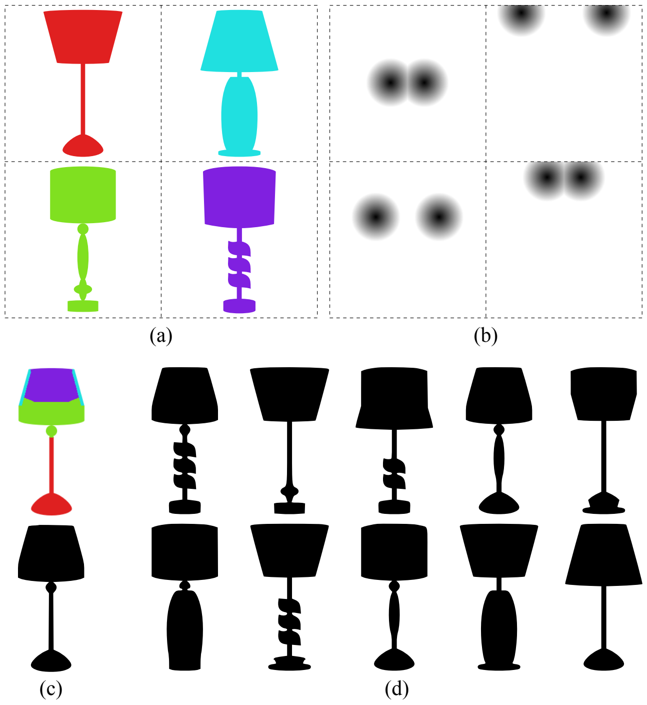

Figure 12: *Pictogram hybrids. Starting from four existing pictograms (a), our system generates random data costs (b) as a guidance for graphcuts, to produce a hybrid pictogram (c). More hybrids are shown in (d). In this example, a horizontal symmetry constraint is enabled.*

4.3. Pictogram Hybrids

Another way of using our system is to automatically synthesize hybrids of a collection of pictograms, inspired by Structured Image Hybrids [RHDG10]. Fig. 12 demonstrates an example. To synthesize hybrids, we randomly select a *seed position* from each exemplar in the collection to retain in the remixing result. In the case of symmetric shapes, we augment this to a pair of symmetric seed positions. We then compute truncated distance fields D_H from these seed positions (Fig. 12b), normalized by the truncation threshold $t = 40$ to ensure a maximum value equal to 1. We use the D_H as the data costs for graph-cuts. Intuitively, the pixels around seed positions have low data cost, and thus tend to be retained in the hybrid. The graph-cuts algorithm finds optimal cuts between these seed positions to remix exemplars.

Since the data cost is randomly generated, we need a strong smoothness constraint to ensure the quality of the remixing result. We find $\lambda_S = 3.0$ to be sufficient for all test cases. Needless to say, randomly generated data cost is not guaranteed to produce perfect results all the time. Also, different random seeds can also yield the same hybrids. Therefore, our results figures demonstrate 8 to 14 results from a larger set produced with 40 random seeds.

Fig. 13 shows a set of face hybrids (see also Fig. 1c). In order to maintain semantics, we enabled the horizontal symmetry constraint. However, in contrast to Structured Image Hybrids [RHDG10], we only require the *source positions of pixels*

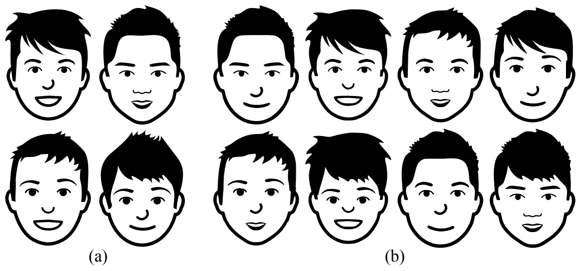

Figure 13: Pictogram hybrids: boy. *(a) existing pictograms; (b) hybrids. A horizontal symmetry cost is enabled, but because the original exemplars have asymmetric hairstyles, the resulting hybrids are allowed to be asymmetric as well. In contrast, the eyes (which are symmetric in the input) are symmetric in the output.*

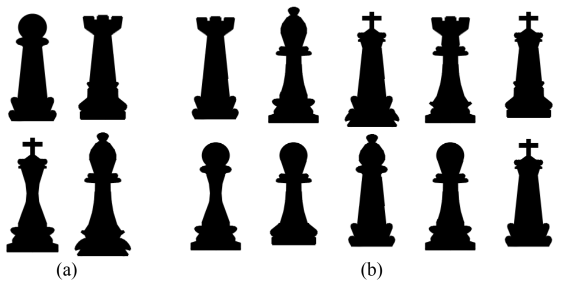

Figure 14: Pictogram hybrids: chess *figures from [RHDG10]. (a) exemplar images; (b) hybrids. A horizontal symmetry constraint is enabled.*

(as opposed to the pixels themselves) to be symmetric. Therefore, our system also retains the asymmetry of hair that is present in the original pictograms. Fig. 14 demonstrates hybrids of the same chess exemplars used in the Structured Image Hybrids [RHDG10] paper. Our system achieves subjectively comparable quality.

4.4. Pictogram Montage

In some cases, users would like more direct control over how exemplars are aligned and remixed. In this scenario, our system resembles a pictogram-optimized instance of Photomontage [ADA*04], which we call *pictogram montage*. Fig. 15 demonstrates an example. We ask the user to select source pictograms, align them, then use strokes to select which parts to retain in the remixing results. We compute truncated distance fields D_M from the pixels covered by the strokes, truncated with threshold $t = 20$ and normalized to a maximum value of 1. We then use D_M as the data cost for graph-cuts. We find that $\lambda_S = 3.0$ works well for all test cases. Fig. 16 demonstrates two more examples.

5. Evaluation

Our primary evaluation is qualitative: we ask the reader to refer to the figures in this paper as well as supplemental materials for examples of all four workflows. In addition, we conducted two studies to evaluate several aspects of our system. First, we evaluate how often our algorithm is able to create a high-quality pictogram from a

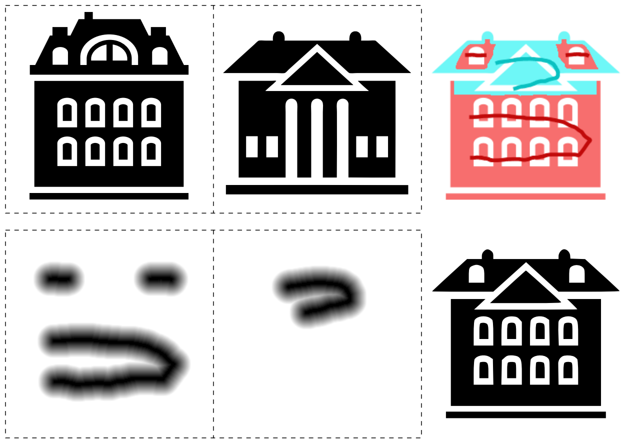

Figure 15: Pictogram montage. The user aligns pictograms of buildings and selects parts with strokes (top row). Our system generates data costs based on the strokes (bottom left), and generates a pictogram montage (bottom right).

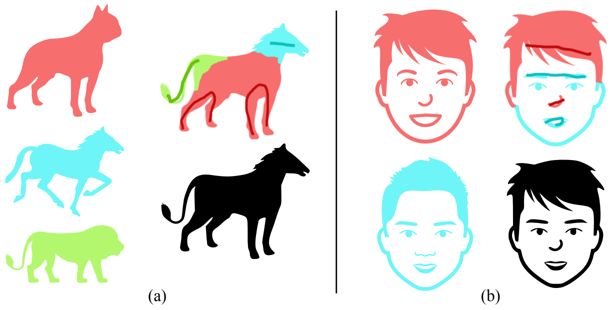

Figure 16: Two more examples of pictogram montage.

user-drawn sketch. Second, we compare the visual quality of algorithmically remixed pictograms against original, hand-drawn icons, to see if humans prefer one over the other.

5.1. A Semi-Blind Test

We first conducted a “semi-blind” test on our sketch-based modeling workflow, in which the authors produced the inputs to the pipeline. There are two reasons why we did this, instead of asking external users. First, we wished to avoid the difficult question of intent, i.e., whether the results match what the users had in mind. Instead, we only evaluate the quality of results. Second, as shown in Sec. 5.3, our system currently is not able to match and remix in real time, which makes it difficult to have users use our system.

For this test, we produced 16 input sketches, covering a diversity of subjects seen in reasonably canonical views. Fig. 17 shows the results of our algorithm on those sketches. Our algorithm produced plausible and recognizable pictograms 75% of the time, and in more than 50% of cases (in the blue box) no existing pictograms matched the sketch polygons (though this determination is necessarily subjective). We include the best match and other details in supplemental materials. Even when there was an existing matching pictogram, it can be valuable to see alternatives; in several cases

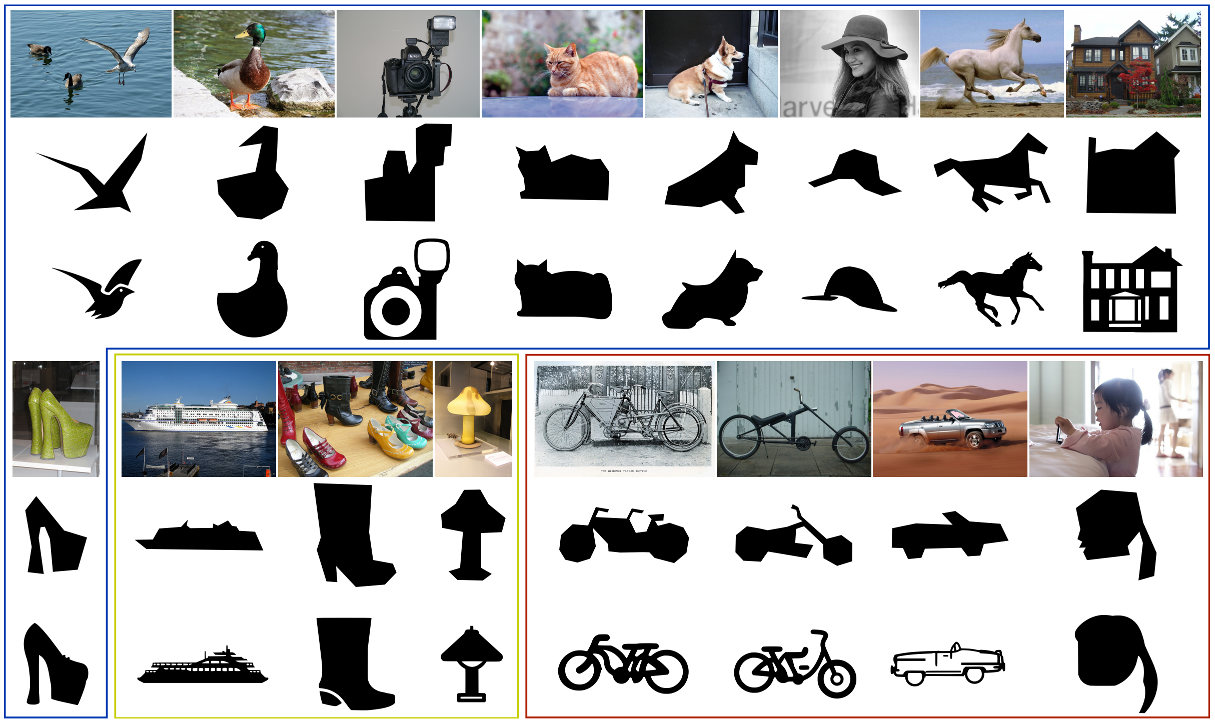

Figure 17: Semi-blind test results. Row 1: photo. Row 2: the user's sketch polygon. Row 3: the best pictogram generated by our algorithm. Blue box: test cases in which no existing pictograms match the user's intent, but our algorithm generates a matching pictogram by remixing. Yellow box: test cases in which our algorithm obtains a reasonable pictogram, but there are existing single pictograms that match the user's intent. Red box: test cases in which our algorithm fails.

(e.g., *boot* and *lamp*) our remixed pictograms have a different appearance or style that the user may prefer.

5.2. User Study on Visual Quality of Results

We also conducted a large-scale human study on Amazon Mechanical Turk to evaluate remixing quality. We choose to evaluate the *pictogram hybrids* workflow, as it is the only workflow without user control, and hence avoids the difficult-to-evaluate questions of user intent mentioned above. We asked each Turker to compare 30 pairs of pictograms for each of three test cases: *boy* (Fig. 1c and Fig. 13), *lamp* (Fig. 12), and *chess* (Fig. 14). We used a forced-choice methodology, in which Turkers had to select their preferred icon within each pair. Among these pairs, 24 of them compare the 4 exemplars against all the results shown in our paper, while the remaining 6 pairs are sanity checks that compare poorly-synthesized pictograms against our results. We obtained the poorly-synthesized icons by using a very low weight (0.001) smoothness cost. Any submission that fails to give a correct answer for any sanity-check pair is discarded. This way, a randomly-guessing user has only a 1.6% chance of passing our sanity check.

Our study received 139 submissions from Turkers in the United States, of which 122 passed the sanity check. Each pair was compared by 20 to 24 users who passed our sanity check. Table 1 shows the results. We observe that about 45% of the time our synthesized icons were preferred, which is close to the 50% that we would expect to observe if the synthesized icons were of completely equivalent quality to the original hand-drawn ones. We thus conclude that, while there is a small difference, it is difficult for users to distinguish between our results and stock pictograms.

Table 1: *Results of user study evaluating visual quality. We show the number of user selections, as well as the percentage, preferring our results vs. stock pictograms.*

Test Case	Ours	Stock	Ours%
boy	512	659	43.7%
lamp	414	514	44.6%
chess	385	444	46.4%

5.3. Running Time

Our algorithm is implemented in Python, with the Fast Fourier Transform implemented via a Python binding for `FFTW` [FJ05]. Graph-cuts are solved via the α-expansion algorithm in the `GCO` package [BVZ01]. For sketch-based workflows, the bottleneck of our algorithm is sliding-window matching. On a 3.6GHz CPU, with a single CPU thread, it takes about 13 msec to match a retrieved pictogram at one scale and reflection for each salient region. Therefore, for 200 retrieved pictograms, 3 different scales, and counting reflections, it takes about 140 seconds to perform sliding-window matching for all 9 salient regions. The multi-label graph-cuts algo-

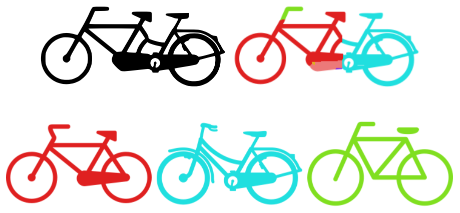

Figure 18: *Remixing result for the tandem bicycle, if we allow a larger set of scaling factors for matching.*

rithm takes 3 to 11 seconds with a single CPU thread, which means that even if we try 8 values for λ_S, the total running time for graph-cuts is less than sliding-window matching. Therefore, in our implementation, after we perform sliding-window matching, we run the graph-cuts algorithm with 8 different values for λ_S, ranging from 0.03 to 0.75, and ask the user to choose the desired result. Both the matching and remixing stages of our algorithm are trivially parallelizable, and we would expect significant speedups with parallel or GPU implementations.

6. Discussion, Limitations, and Future Work

Our algorithm has several failure modes, mainly in sketch-based workflows. First, as a data-driven algorithm, it can fail due to insufficient data. In particular, most artists draw pictograms only in canonical views. If a novice user sketches an object with an arbitrary view, our algorithm might fail. Even given a canonical view, if no existing pictograms partially match the user's sketch polygon, our algorithm will also fail. The "chopper" bicycle in Fig. 17 is an example — it was designed and built by the person who took the photo, different from all normal bicycles. The algorithm will also fail for particular styles (i.e., blackness intervals) if the repository does not provide sufficient coverage for them. We expect that these problems will be gradually alleviated as more and more artists contribute to online repositories of icons.

Second, we expect users to describe the shape of their desired pictogram with a polygon. Because the polygon is rough, we detect salient regions by using large windows, and use TSDF to measure the similarity between the polygon and the exemplars. These technical decisions make our algorithm robust to small changes and noise on the polygon, but on the other hand insensitive to small details. The girl in Fig. 17 is a typical case: the polygon has details to represent the girl's nose and mouth, but our algorithm fails to retain these parts in the result. Giving even higher weight to fine details in the sketch, or allowing users to sketch interior detail, might help alleviate some of these problems.

Third, though modest scaling factors 0.9 and 1.1 are sufficient for sliding-window matching in most cases, there are exceptions. The tandem bicycle case in Fig. 17 is a typical case — it requires an aggressive scaling of wheels to fit two seats in between. We re-ran this example with a more aggressive set of scaling factors (0.8, 0.9, 1.1, and 1.25), giving the correctly remixed result in in Fig. 18. We do not see this as a general solution, though, since adding these additional scaling factors sometimes yields visually implausible results.

References

[ADA*04] Agarwala A., Dontcheva M., Agrawala M., Drucker S., Colburn A., Curless B., Salesin D., Cohen M.: Interactive digital photomontage. *ACM Trans. Graph. (Proc. SIGGRAPH) 23*, 3 (Aug. 2004), 294–302. 2, 3, 6, 9

[BTS05] Barla P., Thollot J., Sillion F. X.: Geometric clustering for line drawing simplification. In *Proc. Eurographics Workshop on Rendering* (June 2005), pp. 183–192. 2

[BVZ01] Boykov Y., Veksler O., Zabih R.: Fast approximate energy minimization via graph cuts. *IEEE Trans. PAMI 23*, 11 (Nov. 2001), 1222–1239. 3, 5, 10

[CCT*09] Chen T., Cheng M.-M., Tan P., Shamir A., Hu S.-M.: Sketch2Photo: Internet image montage. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 28*, 5 (Dec. 2009), 124:1–124:10. 2

[CKGK11] Chaudhuri S., Kalogerakis E., Guibas L., Koltun V.: Probabilistic reasoning for assembly-based 3D modeling. *ACM Trans. Graph. (Proc. SIGGRAPH) 30*, 4 (July 2011), 35:1–35:10. 2

[FJ05] Frigo M., Johnson S.: The design and implementation of FFTW3. *Proc. IEEE 93*, 2 (Feb. 2005), 216–231. 10

[FKS*04] Funkhouser T., Kazhdan M., Shilane P., Min P., Kiefer W., Tal A., Rusinkiewicz S., Dobkin D.: Modeling by example. *ACM Trans. Graph. (Proc. SIGGRAPH) 23*, 3 (Aug. 2004), 652–663. 2

[GAGH14] Garces E., Agarwala A., Gutierrez D., Hertzmann A.: A similarity measure for illustration style. *ACM Trans. Graph. (Proc. SIGGRAPH) 33*, 4 (July 2014), 93:1–93:9. 6

[GCZ*12] Goldberg C., Chen T., Zhang F.-L., Shamir A., Hu S.-M.: Data-driven object manipulation in images. *Computer Graphics Forum (Proc. Eurographics) 31*, 2 (May 2012), 265–274. 2

[HE07] Hays J., Efros A. A.: Scene completion using millions of photographs. *ACM Trans. Graph. (Proc. SIGGRAPH) 26*, 3 (July 2007), 4:1–4:7. 2

[HL12] Hurtut T., Landes P.-E.: Synthesizing structured doodle hybrids. In *SIGGRAPH Asia 2012 Posters* (2012), p. 43:1. 2

[JAWH14] Jun X., Aaron H., Wilmot L., Holger W.: PortraitSketch: Face sketching assistance for novices. In *Proc. UIST* (2014), pp. 407–417. 2

[JTRS12] Jain A., Thormählen T., Ritschel T., Seidel H.-P.: Exploring shape variations by 3D-model decomposition and part-based recombination. *Computer Graphics Forum (Proc. Eurographics) 31*, 2 (May 2012), 631–640. 2

[KCKK12] Kalogerakis E., Chaudhuri S., Koller D., Koltun V.: A probabilistic model for component-based shape synthesis. *ACM Trans. Graph. (Proc. SIGGRAPH) 31*, 4 (July 2012), 55:1–55:11. 2

[KSE*03] Kwatra V., Schödl A., Essa I., Turk G., Bobick A.: Graphcut textures: Image and video synthesis using graph cuts. *ACM Trans. Graph. (Proc. SIGGRAPH) 22*, 3 (July 2003), 277–286. 3

[LBW*14] Lu J., Barnes C., Wan C., Asente P., Mech R., Finkelstein A.: DecoBrush: Drawing structured decorative patterns by example. *ACM Trans. Graph. (Proc. SIGGRAPH) 33*, 4 (July 2014), 90:1–90:9. 3

[LH06] Lefebvre S., Hoppe H.: Appearance-space texture synthesis. *ACM Trans. Graph. (Proc. SIGGRAPH) 25*, 3 (July 2006), 541–548. 4

[LZC11] Lee Y. J., Zitnick C. L., Cohen M. F.: ShadowDraw: Real-time user guidance for freehand drawing. *ACM Trans. Graph. (Proc. SIGGRAPH) 30*, 4 (July 2011), 27:1–27:10. 2

[ML11] Ma T., Latecki L. J.: From partial shape matching through local deformation to robust global shape similarity for object detection. In *Proc. CVPR* (2011), pp. 1441–1448. 2

[OLGM11] Ovsjanikov M., Li W., Guibas L., Mitra N. J.: Exploration of continuous variability in collections of 3D shapes. *ACM Trans. Graph. (Proc. SIGGRAPH) 30*, 4 (July 2011), 33:1–33:10. 2

[RHDG10] RISSER E., HAN C., DAHYOT R., GRINSPUN E.: Synthesizing structured image hybrids. *ACM Trans. Graph. (Proc. SIGGRAPH) 29*, 4 (July 2010), 85:1–85:6. 2, 5, 8, 9

[SFCH12] SHEN C.-H., FU H., CHEN K., HU S.-M.: Structure recovery by part assembly. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 31*, 6 (Nov. 2012), 180:1–180:11. 2

[VH01] VELTKAMP R. C., HAGEDOORN M.: State of the art in shape matching. In *Principles of Visual Information Retrieval*, Lew M., (Ed.). Springer-Verlag, 2001, pp. 87–119. 2

[XZZ*11] XU K., ZHENG H., ZHANG H., COHEN-OR D., LIU L., XIONG Y.: Photo-inspired model-driven 3D object modeling. *ACM Trans. Graph. (Proc. SIGGRAPH) 30*, 4 (July 2011), 80:1–80:10. 2

[Zit13] ZITNICK C. L.: Handwriting beautification using token means. *ACM Trans. Graph. (Proc. SIGGRAPH) 32*, 4 (July 2013), 53:1–53:8. 2

Appendix A: Accelerating Matching with the FFT

We specialize the matching error (4) discussed in Section 3.3 to the case of pure translation, and split it into three terms:

$$\delta(u,v) = \sum_{x,y} M_i(x,y)\big(D_j(u+x,v+y) - D_P(x,y)\big)^2 \tag{12}$$

$$= \delta_1(u,v) - 2\,\delta_2(u,v) + \delta_3(u,v) \tag{13}$$

$$\delta_1(u,v) = \sum_{x,y} M_i(x,y)\,D_j(u+x,v+y)^2 \tag{14}$$

$$\delta_2(u,v) = \sum_{x,y} M_i(x,y)\,D_P(x,y)\,D_j(u+x,v+y) \tag{15}$$

$$\delta_3(u,v) = \sum_{x,y} M_i(x,y)\,D_P(x,y)^2 \tag{16}$$

We may think of δ_1 as filtering D_j^2 with M_i, and δ_2 as filtering D_j with $M_i D_P$. These operations can be accelerated with the Fast Fourier Transform. δ_3 is a constant that has no effect on the relative ordering of matching scores, and may be omitted.