Autocomplete Sketch Tool

Sam Seifert, Georgia Institute of Technology
Advanced Computer Vision
Spring 2016

I. Abstract

This work details an application that can be used for sketch auto-completion. Sketch auto-completion can be used to create sketches faster and increase sketch quality. Users sketch on a canvas, and the application searches for and orients a nearest neighbor to the current canvas contents using the dataset from *How do Humans Sketch Objects* [1]. Users can choose to continue drawing, complete the sketch with the current nearest neighbor, or preview a different nearest neighbor. While this application proved useful, the high variability in sketch quality from *How do Humans Sketch Objects* created scenarios where the artist work was more appropriate than the proposed autocompletes.

II. Introduction

Many of today's smartphones offer autocomplete, a feature that predicts what words the user is going to type before they are going to type them. Autocomplete saves users time, and can improve sentence structure and content. Another content generation task, sketching, could also benefit from an autocomplete tool. Where text autocomplete makes users better spellers, sketch autocomplete could make users better artists. In recent years there has been work in the human sketch space, but there has not been a tool designed to autocomplete sketches. This work is a first attempt at sketch auto-completion.

III. Previous Work

How do Humans Sketch Objects [1] introduces a sketch dataset created through crowdsourcing. The dataset contains 20,000 sketches of 250 object categories, with exactly 80 sketches per category. The authors experiment with sketch classification using the dataset, and achieve a classification accuracy of 55% using a series of 1 vs. all SVM classifiers. *How do Humans Sketch Objects* proposed the first large scale sketch dataset, and the sketches used for auto-completion in this work are taken from their dataset.

Shadowdraw [2] introduces a sketching tool that automatically generates a semitransparent tracing layer behind the drawing canvas. This tracing layer provides queues to the user about object size, spacing, and perspective. When the tool is working, the tracing layer displays roughly what the user intends to draw, and then the user can trace the object to finish the sketch. The tracing layer is generated by matching the current canvas contents against edge images generated from thousands of images taken from the internet. The top matches are then combined (with alpha blending) and placed on the canvas.

The *Shadowdraw* pipeline is designed to help users generate sketches that are faithful to images. However, as indicated by Eitz in [1], humans are not faithful artists. This technique would struggle to generate a meaningful tracing layer when users draw rabbits with exaggerated ears or humans as stick figures. This raises an important question about designing a sketch assist tool: is it more important to maintain the artistic qualities that make a sketch a sketch, or constrain the sketch to appear more like an object in a photograph? The answer is ultimately up to the user, and I think both directions are worth pursuing.

Real-time Drawing Assistance through Crowdsourcing [3] proposes modifying inputs at a stroke level using previous stroke data from other users. This pipeline requires several prohibitive constraints: before meaningful modification data can be generated, many users must sketch the same object from the same perspective at the same size with their software. While these constraints prevent broad use of this technique, the presented results

are interesting. Modifying each stroke in realtime is a powerful capability because each stroke helps define each future stroke. By fixing errors before they can propagate, the software can help complete loops, connect line segments that were intended to be continuous, and filter out noise from input devices. If their approach could be decoupled from their prohibitive constraints, they would be left with an practical sketch assist tool.

IV. Approach

Sketch autocompletion suffers from a solution abundance problem. There are a near infinite number of sketches that are valid representations of a palm tree, but having a computer generate these images is not trivial. While CNN's have been used to generate low resolution images of specific objects [4], we're in the market for high resolution, complete sketches of a broad range of objects. This criteria led us to a nearest neighbor approach: based on the current canvas contents, the software proposes the most similar sketch from a dataset as the autocompletion. This allows us to skirt the *is this a valid proposal?* question, as the dataset can be pruned to only contain appropriate, completed sketches. It also enables the users to constrain the nearest neighbor search to filter by category, a feature that proves to be very helpful later on.

A. Matching Criterion

The sketch data from [1] is segmented by time, and the entire sketch generation process can be recreated and played back. This is important, because the software will be attempting to find matches for incomplete sketches. Given that the query images are incomplete, we've identified two different matching approaches:

1) Sliding / Scaling Bounding Box:
   Compare the query against only the completed sketches in the sketch dataset, trying different orientations and scales until a best match is found. Do this for all images.
2) Sliding Time Window:
   Compare query sketch against what has been drawn during a series of time windows for each completed sketch.

Approach 2 was chosen because we thought it would work better. Approach 1 creates unconstrained parameters like *the set of image scales* and *sliding window stride length*. Approach 2 also creates unconstrained parameters, like *window size* and *the set of time windows*, however these parameters have elegant solutions.

B. Window Size

The auto-complete functionality should be invariant to query scale and translation. While this can be achieved with a specialized descriptor, our application achieved scale and translation invariance through a cropping and resizing preprocessing step. With the *sliding time window* approach, both the query image and the sketch data within the time window are resized to a constant image size (based on the bounding box of sketch content). Choosing the specific image size was a tradeoff between memory size and information loss. A window size that was too small would return meaningless nearest neighbors, and a windows size too large would slow down the application. Window sizes of 15x15, 21x21, and 28x28 were tested. Over this range, changing the image size parameter had little affect on both speed and matching performance, however 28x28 produced slightly more relevant nearest neighbors and was selected. Figure 1 shows three representative sketches from the *How do Humans Sketch Objects*, and the respective cropped and resized query images for each completed sketch.

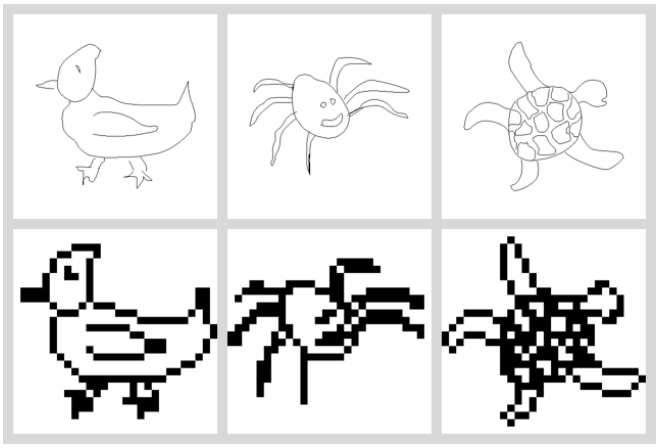

Fig. 1: Completed sketches (top) and their respective cropped & resized query images (bottom).

C. Set of Time Windows

Searching through many time windows, each with a different length or starting point, is an appealing concept. Searching short time windows would help match low level features (eyes of a face, wheels of a car), while searching longer time windows would help match higher level features (face proportions, car outline). However searching many different time windows is intensive, and when the best matching time window comes from the middle of the sketch process it is unclear how to autocomplete the sketch. For example: a user is sketching a bicycle, and the wheel he / she is drawing matches really well to a wheel from a car in the sketch dataset. How should the autocomplete behave? Completing the entire sketch, the car, would be incorrect because the user is drawing a bicycle. Completing just the wheel would be ideal, however the sketch data is not segmented by parts. There is no straightforward way to tell where the wheel ends and the rest of the car begins in the sketch data. Rather than answer these questions, we've constrained each time window to begin at the start of the sketch.

This decision comes with several sacrifices. Draw order is now critical when comparing two images. If the artist of the first sketch draws feature A before feature B, and the artist of the second sketch does the opposite, it is unlikely the two images will match until a point in time where both user's have completed both features. In practice, this isn't as bad as a drawback as it sounds because the order that users sketch the parts of an object is not random (i.e. people tend to draw the windows of a plane after they've drawn the frame). One observation I've made from the *How do Humans Sketch Objects* dataset is that ordering of the subparts of the subparts (not a typo) is even correlated. When people draw the outline of a plane, more often than not, they start with the top, and continue in a clockwise fashion. This holds true for many other oval like (I almost wrote ovular) objects.

Search results have been shown through testing to be robust against the exact number of time windows used for each image. In the examples curated for this paper, the sketch data is segmented into 10 pixel increments, and time windows are generated by how much the user has drawn in terms of pixel length. Each window begins at the start of the sketch: [0, 10] , [0, 20], [0, 30]...[0, n]. Windows that produce duplicate 28x28 descriptors for the same image are then discarded. The remainder of the 28x28 descriptors are stored in a new database. Figure 2 shows the same sketch at five discrete points in time, and the corresponding descriptor for each point in time. Note that the pixel length increment used is dependent on the canvas size for the original sketches. The *How do Humans Sketch Objects* dataset contains only sketches that were drawn on an 800x800 canvas. If the canvas was larger or smaller, a different increment distance would be appropriate, but again, the search results have been shown to be robust against changes in this parameter.

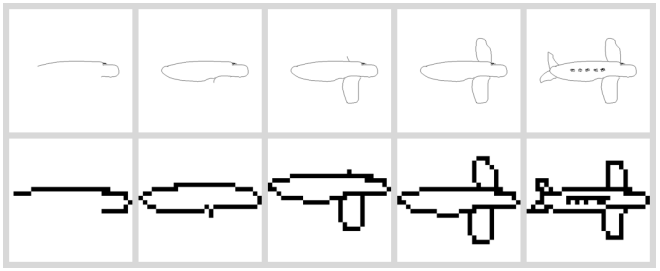

Fig. 2: A single sketch, showed at various stages of completion (top) and the respective cropped & resized query images at each stage (bottom).

Note: the query image in the pipeline is always the cropped and resized version of entire canvas (no time windows). The time windows are only applied to the dataset images that the query image is being compared to.

D. Descriptor

Scale and translation invariance have already been accounted for in a preprocessing step, so the descriptor used to compare two images need not be these things. Intensity invariance, one of the nicer features of the HOG or SIFT descriptors, is also unnecessary, but it is because our dataset consists of only black & white images.

A descriptor with orientation invariance would discard the strong correlation between orientation and content. One of the key findings in *How do Humans Sketch Objects* is that sketches by different artists often have the same perspective

and orientation. Unlike other tasks in Computer Vision, including orientation information creates better matching search results, not worse.

Because invariance to many different transformations was unnecessary, the descriptor used was the raw 28x28 image generated from the crop and resize step, and the comparison function used was a simple squared error summation across the pixels of both images (done twice, once for the original image, and once for a left-right flipped version of the image). Left-right flip invariance was a subset of orientation invariance that, if included, would help generate better matching search returns.

V. Results

Please watch this video of the tool in action: `https://youtu.be/T05qjerd5HM`. In this video, I draw the scene shown in Fig. 3. It contains a sun, two palm trees, two boats, some waves, some birds, and a shoreline. For the sun and only the sun, the category is manually restricted. The query image is just a circle, and to retrieve relevant results, I needed to manually filter out all results that weren't members of the *sun* category. For the rest of the items, no manual restriction is used. The video is a good example of how I envisioned the tool working, the user draws a few strokes to seed the query, and then selects a finished sketch to *autocomplete* the sketch.

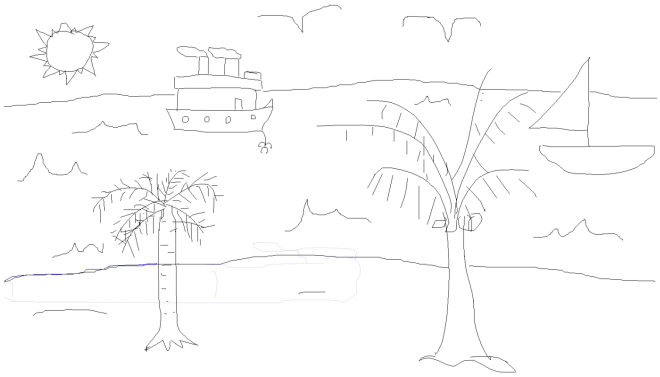

Fig. 3: A scene generated with the autocomplete tool.

A. Autocomplete Tool

The tool performed well at auto completion provided one of two criterion were met:

1) The user had drawn enough content to eliminate search returns that did not match the users intent.
2) The user manually constrained the search results to the category they desired.

If the user did neither of those things, the top search results were dominated by objects from seemingly random categories as shown in Fig. 4. In this example, the query image was taken from the airplane shown in Fig. 2. The best matches are a bowl, a basket, and a submarine. Figure. 5 shows the top search results from the same query image, with the added constraint that all returned objects be airplanes. As you can see, we pull meaningful sketches from the dataset given this constraint, even when the input image is a simple one line curve!

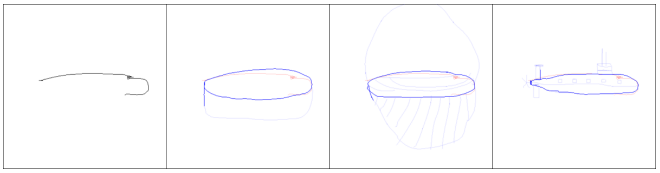

Fig. 4: The top autocomplete suggestions (right), for a query image (left). In the autocomplete suggestion images, the dark blue pixels are the strokes that matched the query, and the light blue pixels are the remainder of the sketch.

An alternative to manually constraining search results to a specific category is to draw enough on the canvas so that items from other undesired categories are no longer top matches. After adding more components to the airplane, the match score for the basket and submarine will drop off, and the match scores from other airplanes will dominate.

B. Sketch Upgrade Tool

One unforeseen use for this tool is as a sketch upgrader. After the user completes a sketch, the software can be used to replace that sketch with a nearest neighbor. The user can draw a rough version of an item (let's say palm tree), and replace it with a better palm tree with identical scale and orientation taken from the database. This is done throughout the video linked above.

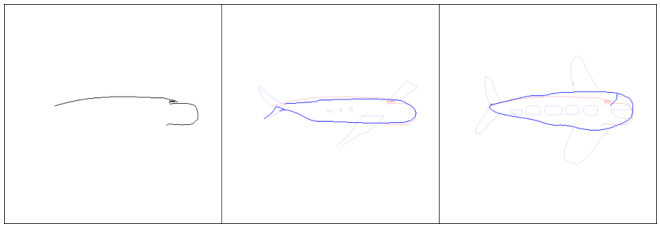

Fig. 5: The top autocomplete suggestions (right), for a query image (left) constraining the results to airplanes only. In the autocomplete suggestion images, the dark blue pixels are the strokes that matched the query, and the light blue pixels are the remainder of the sketch.

VI. Conclusion

This work presents a tool that can be used for sketch auto-completion. Completion is done with a nearest neighbor approach, retrieving sketches that (at some point during the sketching process) resembled the query sketch. The tool works as envisioned, but could be significantly improved with higher quality sketches. An area of future work would be merging the sketched content drawn by the user with a nearest neighbor, instead of just replacing it. As it stands, the tool is a can be used to quickly sketch entire scenes with only a few strokes.

References

[1] M. Eitz, J. Hays, and M. Alexa, "How do humans sketch objects?," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 44–1, 2012.

[2] Y. J. Lee, C. L. Zitnick, and M. F. Cohen, "Shadowdraw: real-time user guidance for freehand drawing," in *ACM Transactions on Graphics (TOG)*, vol. 30, p. 27, ACM, 2011.

[3] A. Limpaecher, N. Feltman, A. Treuille, and M. Cohen, "Real-time drawing assistance through crowdsourcing," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 54, 2013.

[4] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox, "Learning to generate chairs with convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1538–1546, 2015.