DIP FINAL PROJECT REPORT

Project Name: ShadowDraw for freehand drawing

TEAM NAME : fun@images

Team Members: **Ishan Bansal** (201530193), **Swapnil Daga** (201531063)

Introduction

Problem Statement

The aim of this project is to come up with a system for guiding the freeform drawing of objects by giving suggestions based on user drawing. Our project gives the relevant freehand sketches as the user draws in a separate window which can be used as a reference. Apart from providing a single image, we try to *blend several possible images from the dataset* that is possible in order to open up the opportunity for the user to choose among them.

Motivation

ShadowDraw merges artistic talent and creativity of an individual with the precision of computer intelligence. Drawing to precision as such is a *painful task for non-professionals*. People have difficulty handling the ratio of various components and their relative size and shape. People do try to *search for a similar image and trace it but the whole process is a cumbersome task in itself*. It starts from manual searching and finding the perfect image, to tracing it seeing the picture which kills creativity and also is not an accurate representation of the object in many cases. ShadowDraw makes the life of the person drawing an object easy, as it *automatically finds images easily and provides only suggestions* out of which an artist can choose what he wants or go with his own creativity for which he will get suggestions at a later point of time.

Environment Used : OpenCV(3.1) + Python(2.7)

Overview

Our project consists of the following main parts:

1. We *collect images from the web* and build a custom dataset of images. This is done offline.

2. We convert each image to it's edge representation form by using long edge detector technique.

3. We store the overlapping windows in edge images and convert it into edge descriptor form. We then code it as sketches each with a distinct hash key and put it in an inverted hash table.

4. We match the user's imput image with database made above using similar encoding for user's input image and getting it's hash function. We then compare the hash functions which makes the matching fast and display the outputs with top confidence scores.

Algorithm

Part-1 (Dataset Generation)

1. *Take the images* as input.
2. *Extract edges* out of the images using Canny.
3. Divide the Output images into *patches of 60*60* with an *overlap of 50%*.
4. Apply Edge descriptor for each of the input patches.
5. Creating min-hash function by *randomly permuting the descriptor and get the first 1* coming in the descriptor, doing this k times decreases the probability of matching by match^k.
6. Doing this *N-times* so that the corresponding patch has a probability of matching increasing by N.
7. Creating the *Inverse hash table* for the corresponding Patches.

Part - 2 (Image Matching)

1. Take the input image and divide it into *patches of 18*18* with an overlap of 75%.
2. Vote for each of the image and the patch by going over the min-hash function.
3. Take the images having the highest number of matches.
4. These are then weighed depending on the closeness of these with respect to the *Jaccard similarity*.
5. The final edges are drawn on the GUI.

Approaches

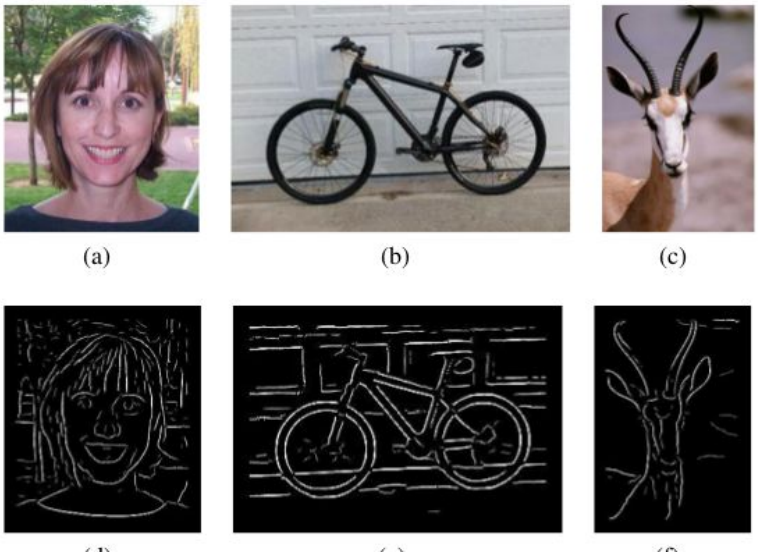
ShadowDraw includes three main components:

1. The construction of an inverted file structure that indexes a database of images and their edge maps.
2. A query method that given user strokes retrieves matching images and weights them based on a matching score.
3. The user interface, which displays weighted edge maps beneath the user's drawing to help guide the drawing process.

Database Creation :

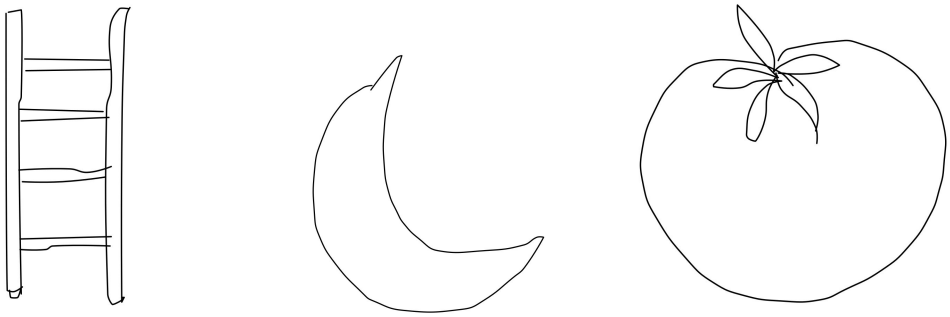
The images in the database should be selected so that they appear as similar to the user as possible. The *images for the dataset are handmade images* different from the one used in the original paper, the reason is a number of extra edges come into picture while using them which lead to false detection. The image dataset has images via approximately 250 categorical queries such as "moon", "bicycle", "ladder", etc. We scale the images to *fit a 300x300 pixel resolution*, i.e., the long side is scaled to 300 pixels. We then process each image in three stages and add them to an inverted file structure. First, *edges are extracted from the image* then *edge descriptors are computed*. Finally, sets of concatenated hashes called sketches are computed from the edge descriptors and added to the database. We store the database as an inverted file, in other words, indexed by sketch value, which in turn points to the original image and its edges.

Images are taken from the dataset which is generated by the Research paper.



(a) (b) (c)

As it is seen that *a number of extra edges* are there in the dataset of the paper which leads to the false prediction or more number of edges in the output shadow therefore in order to prevent this thing from happening the handmade dataset is used, sample images are the following :



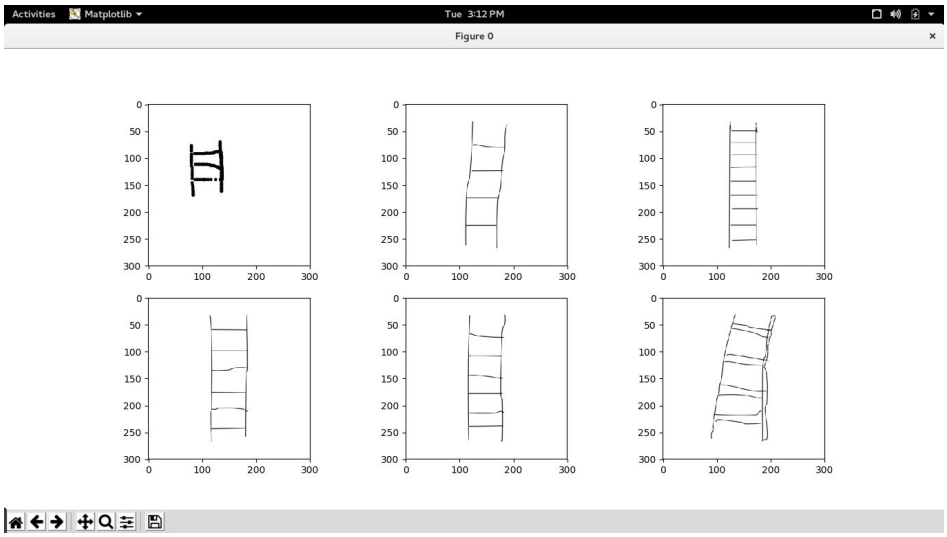
Edge Extraction & Patch Description:

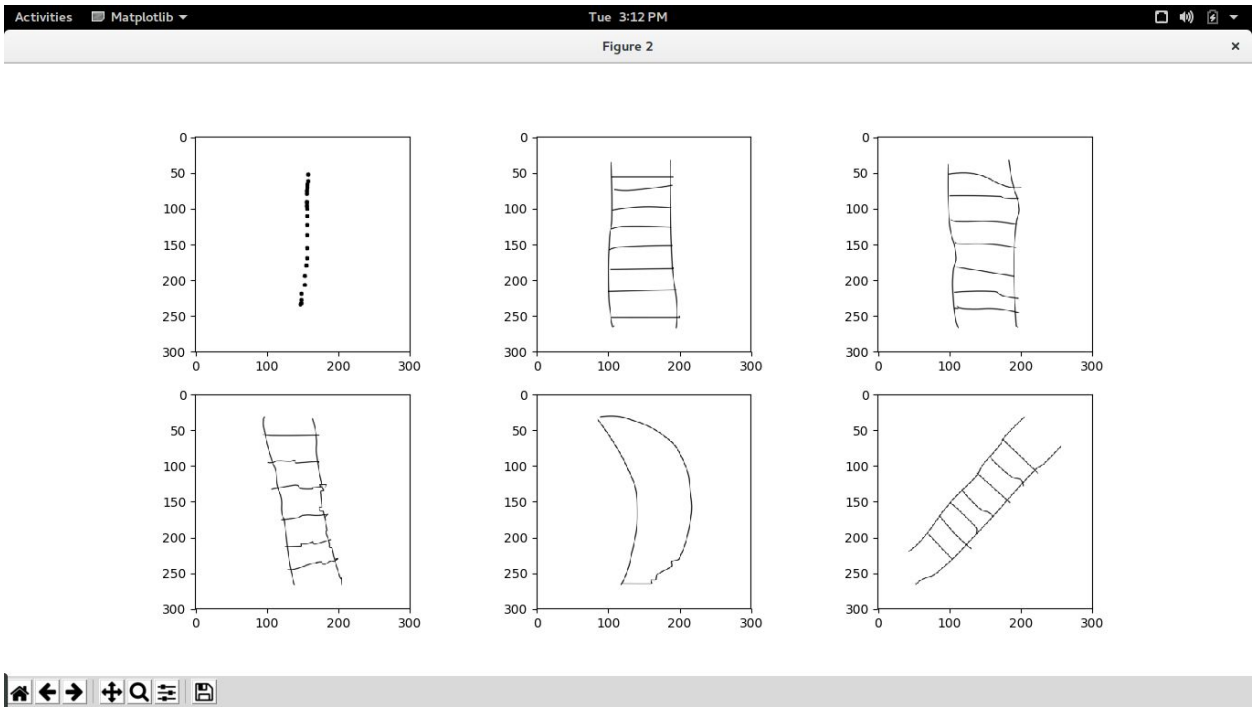
We take the maxima of all perpendicular responses provided by Canny edge detection to find the edge positions for each edge image.. Given an image I in the database with corresponding edges E and orientations θ, we compute a set of edge descriptors. Since the goal is to match an edge image E to incomplete and evolving drawings. We compute the descriptors *locally over 60x60 patches*. The patches used to compute neighboring descriptors *overlap by 50%*. The descriptor which we use for the purpose is *SIFT and KAZE descriptors*. Both of them are used and tested for this purpose and the results of both of them are compared as to see which one will work better in this case. Apart from this two we also try to implement the *BICE descriptor* which is mentioned in one of the papers, but the performance seems to be not as good as the above.

Results over different Images using SIFT and KAZE descriptor

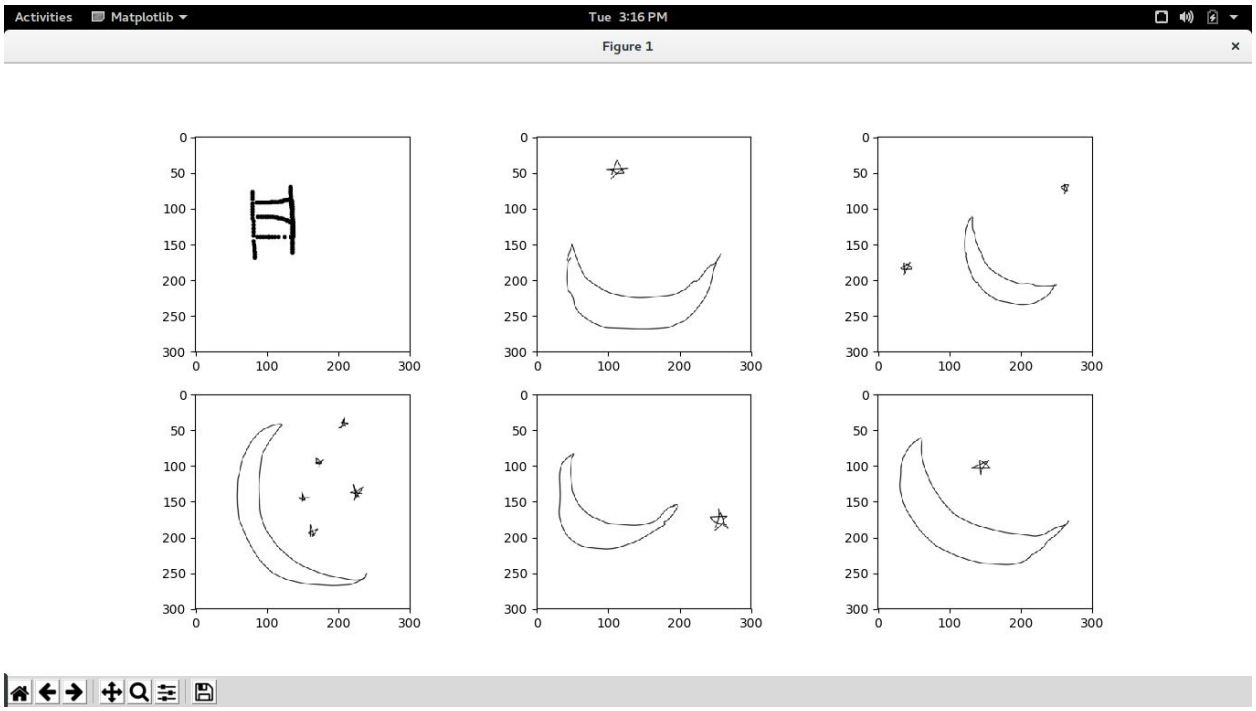
Case1: Partial Ladder Image (KAZE works better)

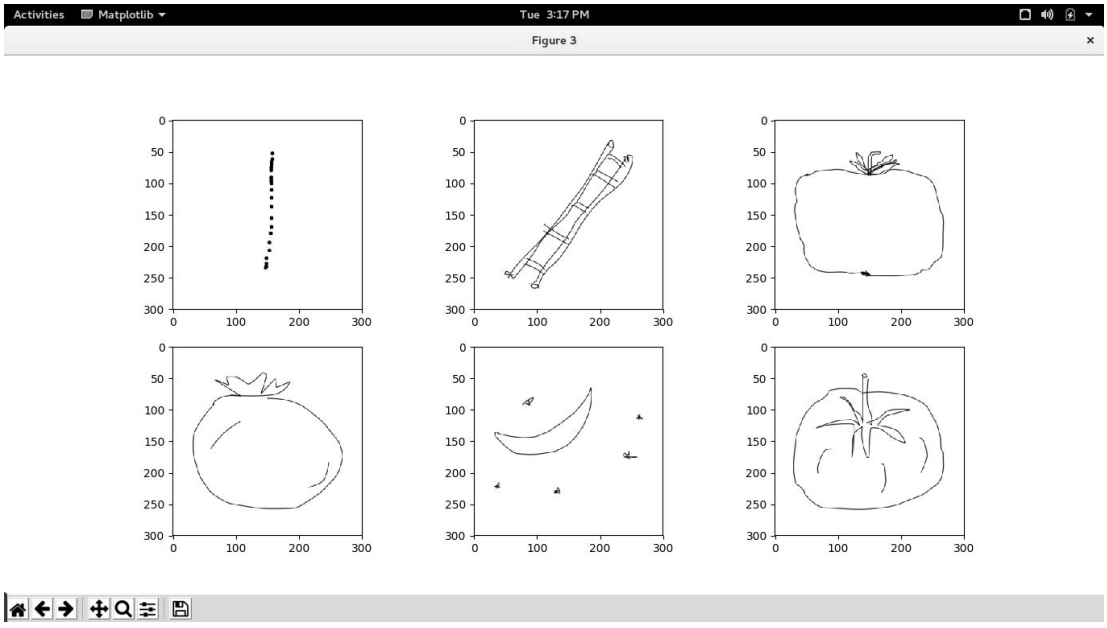
Output of KAZE





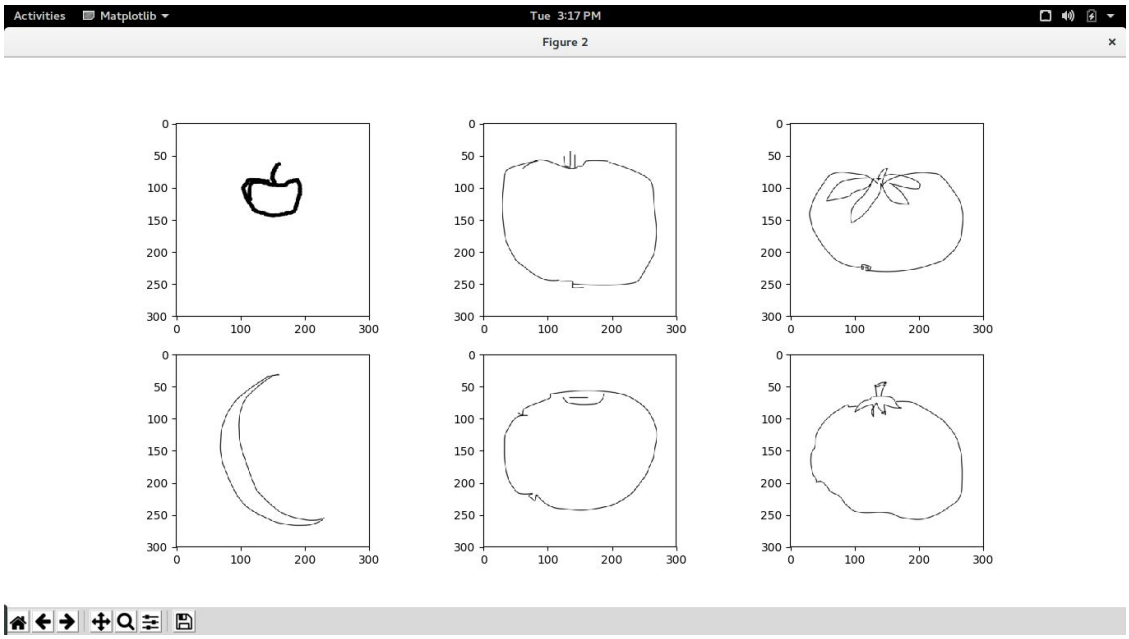
Output of SIFT





Case2: Tomato (SIFT performs better)

SIFT



Min-hash

A min-hash function randomly permutes the set indices (i.e., the ones and zeros). All sets are permuted using the same min-hash function. The min-hash value for a given permuted set is its *smallest index containing a value* of one after the permutation. A single min-hash can be non-discriminative and introduce many false positive retrievals, especially if the descriptor is non-sparse. To increase precision, we compute k independent random min-hash functions and apply them to all descriptors. We concatenate the resulting k min-hash values for each descriptor into *k-tuples called sketches*. The probability of two sketches colliding is thus reduced exponentially to sum(di, dj)^k. To increase recall, this process is repeated n times using *n different sets of k min-hash functions*, resulting in n sketches per descriptor. Trade off will be made between the number of sketches stored for each descriptor and the size of the sketch, k. In our experiments, we store n = 20 sketches of size k = 3 for each descriptor. Here the similarity between the two descriptors is given by the *Jaccard similarity* as the intersection of the cardinality divided by their union.

Image Matching

Image matching is done in two steps:

1. Use of *inverted file structure* to obtain a set of candidate matches.

2. Scoring of candidate matches.

This two step makes the image matching computationally efficient by reducing the number of images to be compared to a very small subset.

Candidate Matching :

We represent the user's drawing as a set of vectorized multi-segment strokes. We create an edge image E from these strokes by drawing lines with a width of one pixel between the stroke points. The rendered lines have the same style as the edges extracted from the natural images in the database, i.e., the edge image E used for matching does not use the stylized strokes. Next, we compute descriptors and their corresponding sketches in the same manner, this time using a *higher resolution grid of 18 × 18* = 324 patches with *75% overlap between neighboring patches.* We use a higher resolution grid to increase the accuracy of the predicted position and to increase invariance to translations in the drawing. Drawing occupies an area of *480 × 480 pixels*, resulting in 96 × 96 pixel patches with 24-pixel offsets between neighbouring patches. We compute descriptors and sketches for each of the 324 patches.

Using the inverse lookup table, we match each sketch from the user's drawing to the sketches stored in the database. A matching sketch casts one vote for the corresponding database image and patch offset pair. We aggregate the matches in a histogram H storing

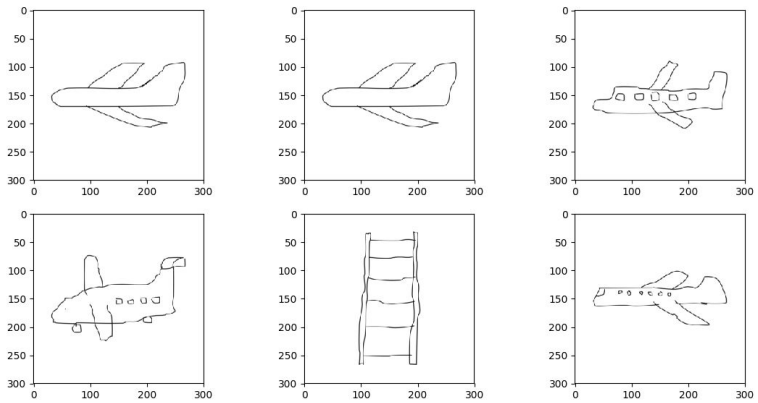
the number of matching sketches for each image at each grid offset. After adding all the matches for each sketch to the histogram, we find the best matching offset for each image, and add the top 5 images to the candidate set C. We compute n = 20 sketches for each descriptor, resulting in a maximum possible 20 votes per sketch in the histogram.

Results

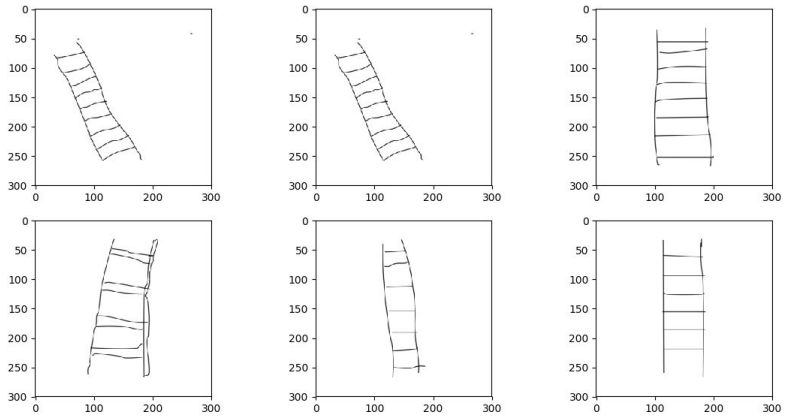
Test1: Taking Random Images from the Dataset and Match with Closest Pairs

Image 1: Input, Rest all are output based on the decreasing order of similarity. (Manually verified)

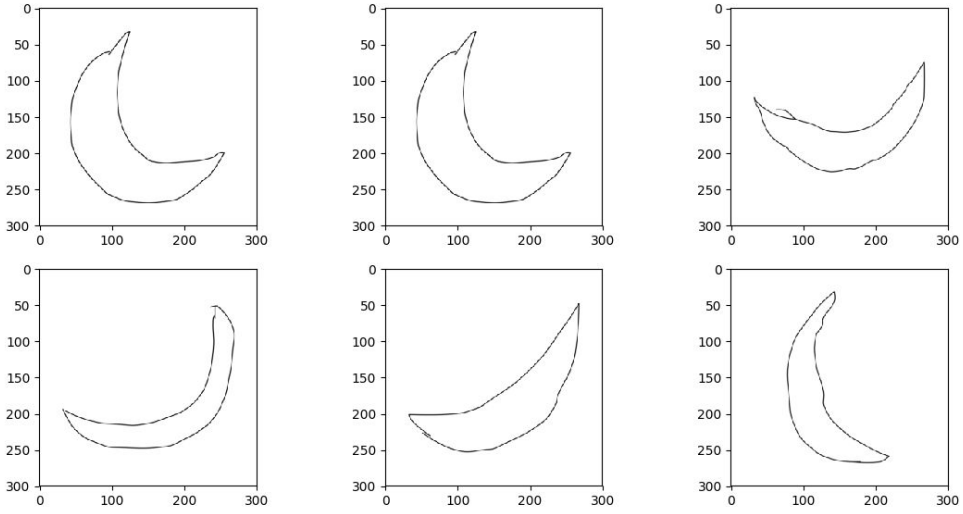
Object 1: Aeroplane



Object 2: Ladder

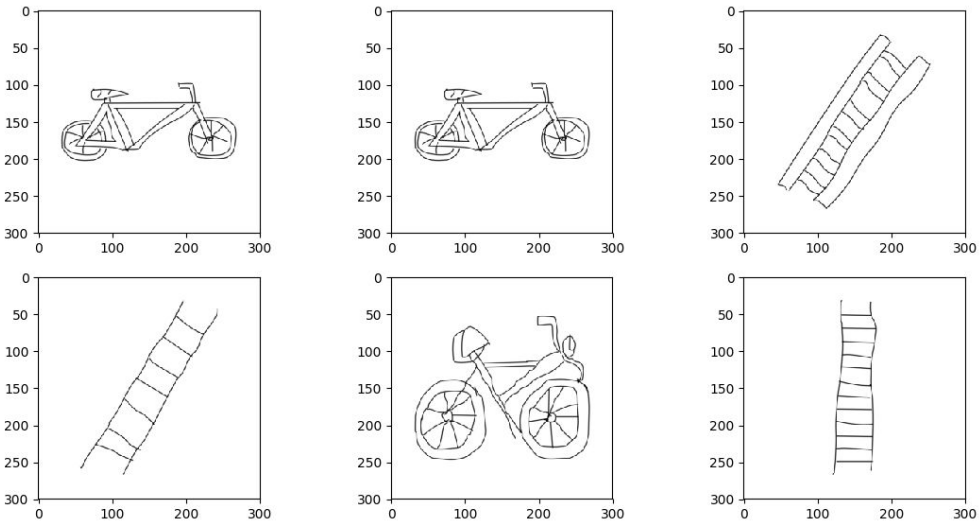


Object 3: Moon



Failure (Bicycle)

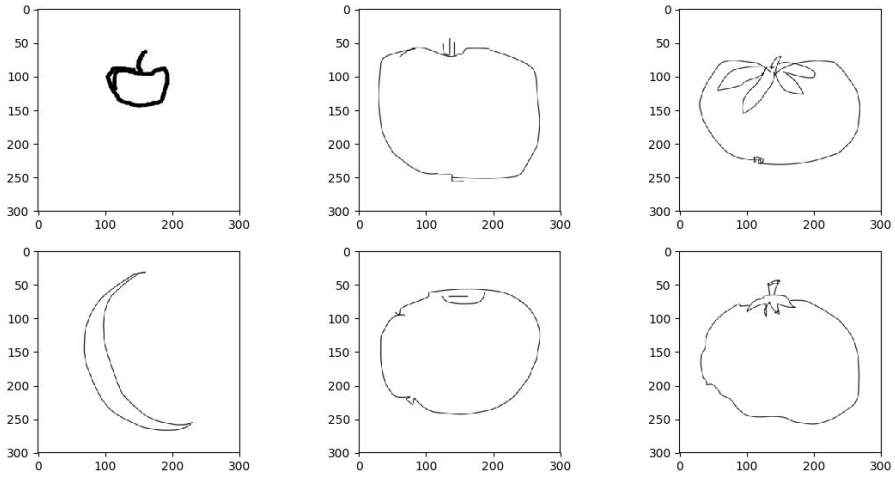
Possible Reason: The Parallel lines are been detected out of the bicycle and are matched with the ladder leading to the following result.



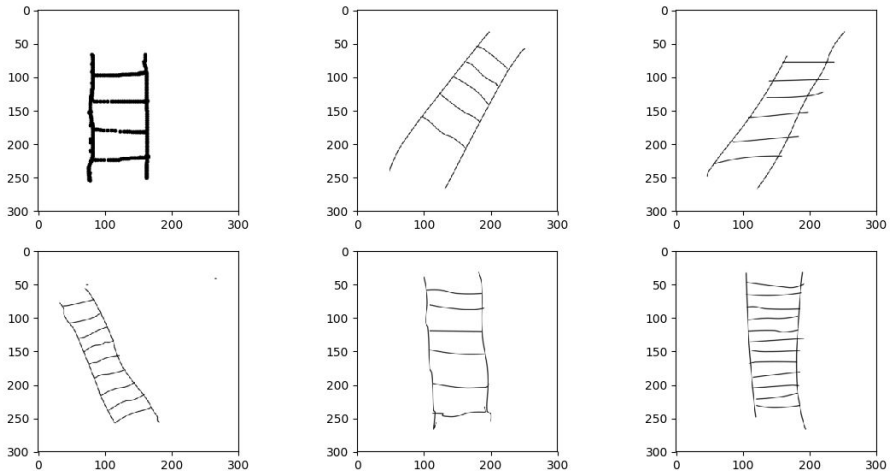
Test2: Draw full images and Compare with the closest ones in the Dataset

Bold one: input, Rest output depending upon the decreasing rate of matching

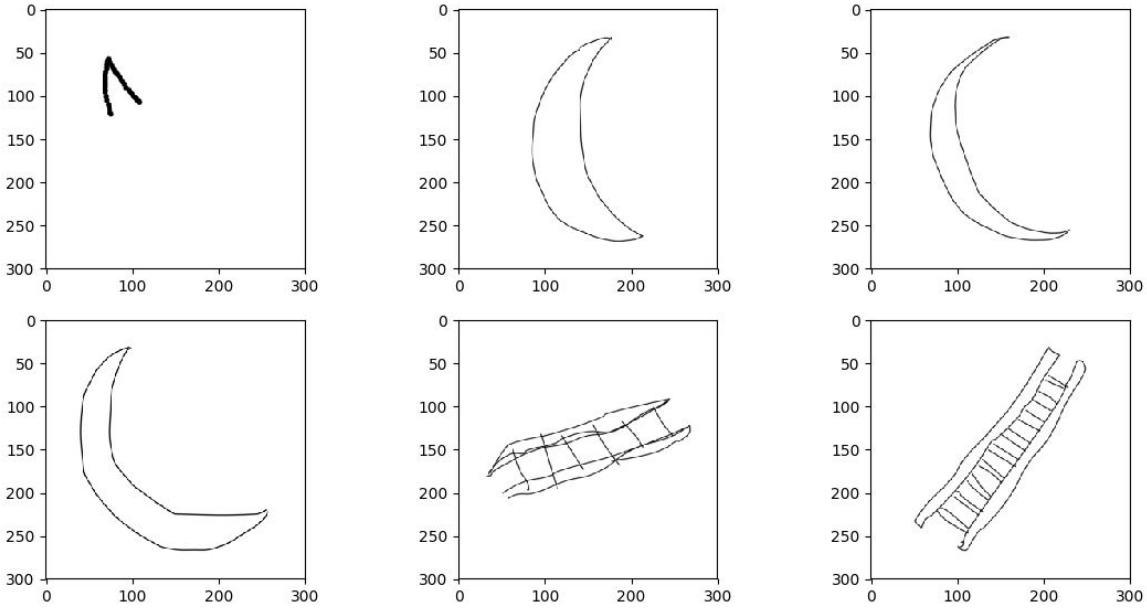
Object1: Tomato



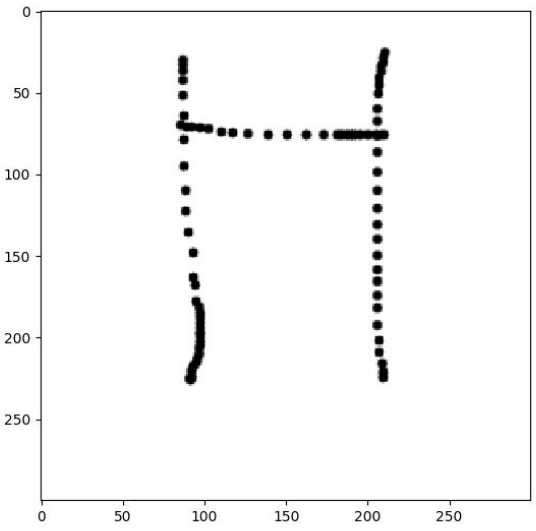
Object2: Ladder

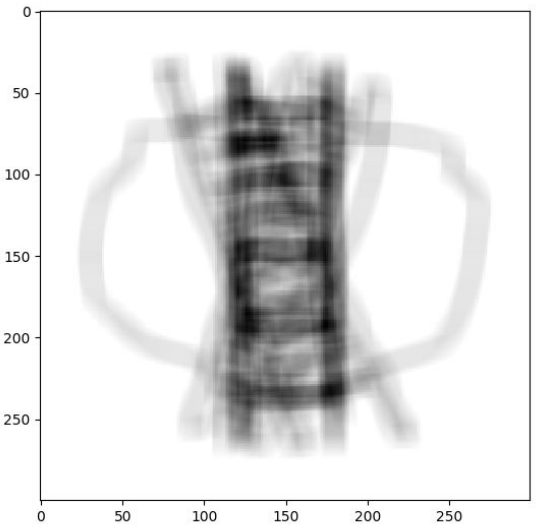


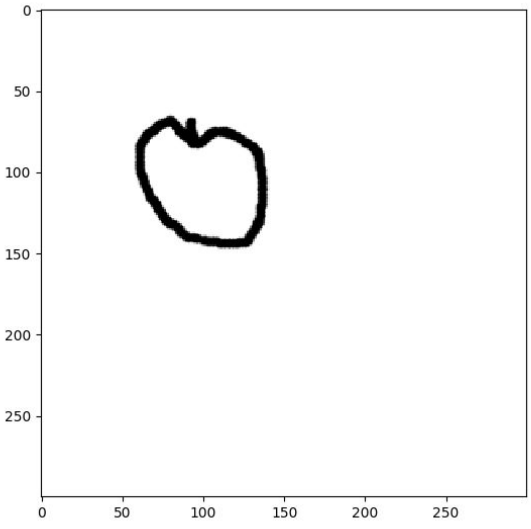
Test3: Partial Images and the closest of all the images in the Dataset

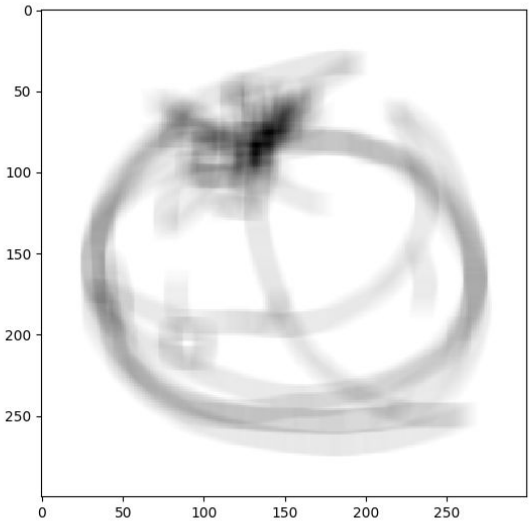


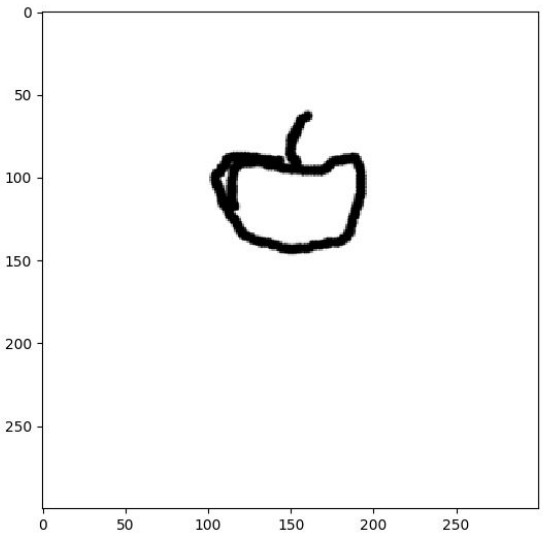
Test4: Drawing shadow out of these images with the corresponding weights

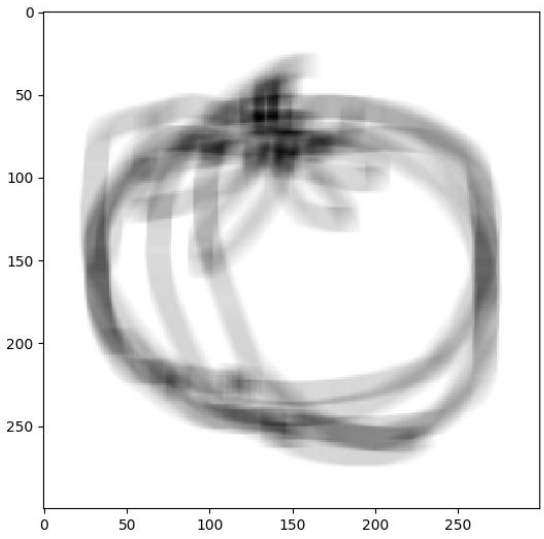

0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250





Discussion

Limitations :

User Interface: The interface can be made better such that the output (shadow) and the input (user drawing) can run on the same screen, in the present both run on different. However, this can be seen as a benefit for some since some people found it frustrating to have a shadow behind their drawing, therefore, putting it over a different screen is a kind of helpful for some.

Rotation Image: Detection of the similar images even undergone rotation can be done by using SIFT as SIFT is rotation invariant however once it is done the aligned of the Images with the one made by the user is a problem with the current code, which can be improved further.

Storing feature vectors: In the current code everytime you run the code it will calculate the bag of the words for the dataset, rather this can be offline once therefore once the input comes only the comparison should take place saving a lot of time and number of computations.

GitHub Link

https://github.com/ishan98/DIP_ShadowDraw

Acknowledgement :

We would like to express our special thanks of gratitude to our guide Prof. Ravi Kiran & Prof. Rajvi Shah for giving us this golden opportunity to do this wonderful project related to Image Processing on the topic (ShadowDraw for freehand drawing), which helped us in doing a lot of Research and we came to know about many new things . We are really thankful to our mentor Ashwin Pathak for providing us valuable guidance and resources which helped a lot in the completion of this project.

References

- https://www.researchgate.net/publication/220183763_ShadowDraw_Real-Time_User_Guidance_for_Freehand_Drawing (Research Paper)
- https://pythonprogramming.net/ (Opencv python tutorials)
- http://docs.opencv.org/3.1.0/d6/d00/tutorial_py_root.html (Opencv 3.1python tutorials)
- http://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/
- http://docs.opencv.org/3.1.0/d7/d9f/tutorial_linux_install.html