CGM lab

Project A - Spring 19

Drawing with a camera

Layan Jarjoura [redacted]

Ayham Haj [redacted]

Supervides by: Ori Nitzan

Spring, 2019

Contents

Project Goal

Our goal is to tackle an issue that concerns every animator, graphic designer and digital artist. It is to make an accurate substitute for graphic design tools and tablets which is also affordable and available-for-all because the available ones in the market can cost up to hundreds and thousands of dollars currently.

Existing Solutions

A quick research in the graphic design tools market shows that the leading companies who sell tablets and smart-pads don't offer affordable ones.

The good ones – on one hand - work accurately and are easy to use. On the other hand, they usually cost around ten thousand shekels and should always be carried with one-self and taken care of.

An example of an 800$ touch pen display from Wacom:

The features they provide include:

- User-customized drawing.
- Drawing shapes.
- Picking different colors
- Showing the drawing live on a computer screen that is connected to the tablet.

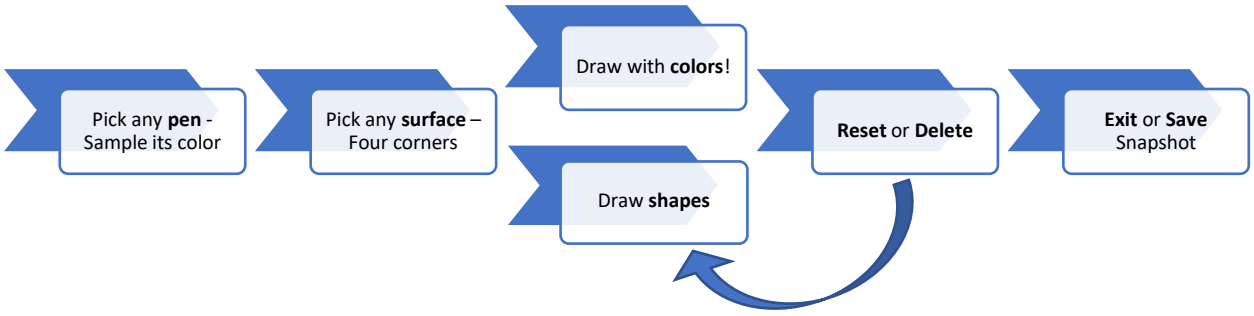
Chosen Solution

The general idea is to allow the user to use his regular computer camera and a regular pen and our project will turn those into a drawing 'tablet'.

Our program recognizes the pen firstly, and then turns the chosen surface into a 'canvas' for digital drawings and doodles.

That way, we saved all the money invested into making 'smart' tablets and pens since we only need a camera and a regular pen. Also, we managed to make graphic design available for a wider audience of artists and talented but less fortunate youngsters.

A general block diagram of how to use it:



Detailed steps for using it:

- We chose to allow the user to:

  1. Pick <u>whatever pen</u> he has laying around (can be a regular blue pen for example) and thus saving the user from paying hundreds of shekels for a tablet-suitable pen which also increases our tool's availability.

  How?

  The user shows the pen he chose to the camera, our program takes samples of its **<u>color</u>** and uses them to detect the pen throughout the drawing process using computer vision and image processing techniques.

  2. Pick <u>whatever surface</u> he wants to draw on. It can be a table/paper/another screen/etc. That would influence the price the most – saving thousands of shekels on a specific-purpose application tablet.

  How?

  The user picks 4 coordinates to define the **<u>corners</u>** of the surface he is willing to draw on. The camera recognizes those corners and performs a geometrical transformation to turn them into a white canvas and show it on the screen.

3. Simply start drawing. The user can customize a random free drawing or, alternatively, draw functions, shapes and more.

How?

We implemented an algorithm that draws up to 4^{th} polynomial functions, plus shapes - such as circles and rectangles. In addition, different **colors** are available for use.

4. An all-while computer screen - resembling a "paper" - shows the drawing live on the user's computer screen.

How?

We implemented an algorithm that performs a geometrical transformation for every frame it captures (the camera captures **frames in a loop**) so that the white canvas on the screen is in the right direction no matter the angle that the user draws in.

- While working on the solution we realized that by picking the surface to be any computer screen, we're technically converting any normal screen to a touch screen.

Project Development Process

The project consisted by the following steps:

- We started the project by taking some time to learn and/or hone our Python programming skills. After that, we did a little research on relevant computer vision Python packets and open-source related projects to get an idea on how to start our own.
- We -then - planned the basic version of the project which included the basic functions: drawing with one color, on a white canvas, in medium accuracy. In addition, a more complex, extensive version was laid out.
- Started programming and testing each step of the way:
  - Capturing a video from the camera in a continuous loop.
  - Detecting pen color using a few samples taken by the camera.
  - Detecting pen head and movement using an algorithm which detects the coordinates where the shadow of the pen contacts the surface.
  - Picking a surface and performing a geometrical transformation to match the camera angle.
  - Drawing the pen movement traces – we later added a color palette to choose from.
  - For the extensive version, we made drawing up to 4^{th} degree polynomial functions and a few shapes (e.g. Circles and rectangles) available.
- Added features such as an eraser, a reset button, a user-friendly gui (Graphic User Interface) and the option to save the drawing as a jpg file.
- Polishing our shadow detection and color detection code to make them more accurate to suit the extensive version vision.
- Final testing in different environments (room, class, lab, different lightning, etc.).

Background information

Mathematical morphology (MM):

MM is a theory and technique for the analysis and processing of geometrical structures, based on set theory, lattice theory, topology, and random functions. MM is most commonly applied to digital images, but it can be employed as well on graphs, surface meshes, solids, and many other spatial structures.

Dilation:

Dilation (usually represented by $\oplus$) is one of the basic operations in mathematical morphology. Originally developed for binary images, it has been expanded first to grayscale images, and then to complete lattices. The dilation operation usually uses a structuring element for probing and expanding the shapes contained in the input image.

Erosion:

Erosion (usually represented by $\ominus$) in morphological image processing from which all other morphological operations are based. It was originally defined for binary images, later being extended to grayscale images, and subsequently to complete lattices. The basic effect of the operator on a binary image is to erode away the boundaries of regions of regions of foreground pixels (*i.e.* white pixels, typically). Thus, areas of foreground pixels shrink in size, and holes within those areas become larger.

Opening and closing:

Opening and closing are two important operators from mathematical morphology. They are both derived from the fundamental operations of erosion and dilation. Like those operators they are normally applied to binary images, although there are also graylevel versions. The basic effect of an opening is somewhat like erosion in that it tends to remove some of the foreground (bright) pixels from the edges of regions of foreground pixels. However, it is less destructive than erosion in general. As with other morphological operators, the exact operation is determined by a structuring element. The effect of the operator is to preserve *foreground* regions that have a similar shape to this structuring element, or that can completely contain the structuring element, while eliminating all other regions of foreground pixels.

Important Code Sections

- Detecting the object's shadow (pen shadow) and its lowest point (pen head):

We used binary masks, morphological operations and contouring in order to find the lowest point of the pen's shadow (the head of the shadow).

```
def get_shadow_min(image,x0,x1,y0,y1,lower,upper):
    cropped_image = image[y0:y1, x0:x1]
    tmp_img = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2HSV)
    cropped_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)
    output = cv2.inRange(tmp_img, lower, upper)
    re, mask_shadow = cv2.threshold(cropped_image, 80, 255, cv2.THRESH_BINARY_INV)
    closing = mask_shadow - output
    kernel = np.ones((5, 5), np.uint8)
    img_dilation = cv2.erode(closing, kernel, iterations=1)
    closing = cv2.morphologyEx(img_dilation, cv2.MORPH_CLOSE, kernel)
    img_dilation = cv2.dilate(closing, kernel, iterations=1)
    closing = cv2.morphologyEx(img_dilation, cv2.MORPH_CLOSE, kernel)
    cv2.imshow('cropped', closing)
    shadow_contours, im2 = cv2.findContours(closing, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    y = 0
    topmost_shadow = (0, 0)
    for shape in shadow_contours:
        tmp = tuple(shape[shape[:, :, 1].argmax()][0])
        if tmp[1] > y:
            y = tmp[1]
            topmost_shadow = tuple(shape[shape[:, :, 1].argmax()][0])
    return topmost_shadow
```

- Detecting the lowest point of the pen – its head:

```
def get_color_min_point(image,lower,upper):
    tmp_img = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    output = cv2.inRange(tmp_img, lower, upper)
    # Taking a matrix of size 5 as the kernel (for erosion and dilation)
    kernel = np.ones((5, 5), np.uint8)
    # The first parameter is the original image,
    # kernel is the matrix with which image is
    # convolved and third parameter is the number
    # of iterations, which will determine how much
    # you want to erode/dilate a given image.
    img_dilation = cv2.dilate(output, kernel, iterations=1)
    kernel = np.ones((10, 10), np.uint8)
    closing = cv2.morphologyEx(img_dilation, cv2.MORPH_CLOSE, kernel)
    contours, im2 = cv2.findContours(closing, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    y = 0
    topmost = (100, 100)
    for shape in contours:
        tmp = tuple(shape[shape[:, :, 1].argmax()][0])
        if tmp[1] > y:
            y = tmp[1]
            topmost = tuple(shape[shape[:, :, 1].argmax()][0])
    return topmost,closing
```

Using the above two points, we compare them to recognize when the pen's head contacts the paper. When that happens, it means that the user is willing to draw now and consequently, the program follows and draws its trace.

- Picking the 4 corners of the surface (canvas):

```
pressed_key = cv2.waitKey(1)
if pressed_key & 0xFF == ord('c'):
    points.append(topmost)
    print("point "+str(count)+" choosen")
    count=count+1
if count ==5:
    tmp_point=points[3]
    points[3]=points[2]
    points[2]=tmp_point
    do_once=1
    warped, M = fix_paper.fix(image, points)
    wraped_white, M = fix_paper.fix(white_pic_org, points)
    dims = warped.shape
    height = int(dims[0])
    width = int(dims[1])
    gui = App(tkinter.Toplevel(), "hg", width, height)
tmp_image=image
cv2.circle(tmp_image, topmost, 10, blue, 3)
cv2.imshow("tmp_image", tmp_image)
traj = np.array([], np.uint16)
traj = np.append(traj, topmost)
dist_pts = 0
dist_records = [dist_pts]
func_points = np.array([], np.uint16)
func_points = np.append(func_points, topmost)
dist_records_func=[dist_pts]
```

The user picks 4 points in the shape of a rectangle by pointing the pen at the chosen corner and pressing 'c' on the keyboard. Repeat 4 times. The program takes the 4 points that can be in any order (no specific order required), re-orders them and sends them to the function *fix_paper.fix* which performs a geometrical transformation and transforms the canvas that's in a weird angle to a rectangular, straight, white canvas on the screen.

- Transforming the 4 corners into a straight screen: (after re-ordering the 4 coordinates)

```
def four_point_transform(image, pts):
    # obtain a consistent order of the points and unpack them
    # individually
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
    # compute the width of the new image, which will be the
    # maximum distance between bottom-right and bottom-left
    # x-coordiates or the top-right and top-left x-coordinates
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))
    # compute the height of the new image, which will be the
    # maximum distance between the top-right and bottom-right
    # y-coordinates or the top-left and bottom-left y-coordinates
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    # now that we have the dimensions of the new image, construct
    # the set of destination points to obtain a "birds eye view",
    # (i.e. top-down view) of the image, again specifying points
    # in the top-left, top-right, bottom-right, and bottom-left
    # order
    dst = np.array([  [0, 0],    [maxWidth - 1, 0],      [maxWidth - 1, maxHeight - 1],
        [0, maxHeight - 1]], dtype = "float32")
    # compute the perspective transform matrix and then apply it
    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
    return warped,M # return the warped image
```

The function which actually performs the geometrical transformation.

- Detecting pen color:

```
def pen_color(frame):
    global hand_rect_one_x, hand_rect_one_y
    blue_cnt=0
    green_cnt = 0
    red1_cnt = 0
    red2_cnt = 0
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    #creates
    roi = np.zeros([70, 10, 3], dtype=hsv_frame.dtype)

    for i in range(total_rectangle):
        roi[i * 10: i * 10 + 10, 0: 10] = hsv_frame[hand_rect_one_x[i]:hand_rect_one_x[i] + 10,
                                          hand_rect_one_y[i]:hand_rect_one_y[i] + 10]
    for i in range(len(roi)):
        for j in range(len(roi[i])):
            if roi[i][j][0] >= blue_lower[0] and roi[i][j][0] <= blue_upper[0] and roi[i][j][1] >= blue_lower[1] and roi[i][j][1] <
                blue_cnt=blue_cnt+1
            if roi[i][j][0] >= green_lower[0] and roi[i][j][0] <= green_upper[0] and roi[i][j][1] >= green_lower[1] and roi[i][j][1
                green_cnt=green_cnt+1
            if (roi[i][j][0] >= red_lower1[0] and roi[i][j][0] <= red_upper1[0]) and roi[i][j][1] >= red_lower1[1] and roi[i][j][1
                red1_cnt=red1_cnt+1
            if (roi[i][j][0] >= red_lower2[0] and roi[i][j][0] <= red_upper2[0]) and roi[i][j][1] >= red_lower2[1] and roi[i][j][1
                red2_cnt=red2_cnt+1
    return maximum(blue_cnt,green_cnt,red1_cnt,red2_cnt)
```

We defined a range of RGB numbers for each color. After sampling the pen's color through the camera, we find the range it belongs to and thus determine its color to track it throughout the drawing. The range it belongs to is determined by the majority of points (which belong to the same range-defined color).

Future development

We have a few suggestions about future developments:

1. This project can serve many purposes other than our own. It can be adjusted to fit other applications such as a math board for blind people, a fun drawing app for kids or a creative platform for artists when adding the style transfer feature.
2. This project can definitely lay a groundwork for future projects in the faculty that serve different purposes as mentioned previously.
3. We are planning to upload it as an open-source code for others to use and benefit from just like we used other GitHub projects for inspiration and ideas.
4. Investing in more accurate algorithms in difficult environments (bad lighting, pens with hard-to-detect colors, etc.).

The End