Application of Computer Vision to Project Sewing Patterns onto Fabric

Michelle Zhuang, '23

Follow this and additional works at: https://works.swarthmore.edu/theses

Recommended Citation
https://works.swarthmore.edu/theses/302

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License

Please note: the theses in this collection are undergraduate senior theses completed by senior undergraduate students who have received a bachelor's degree.
This work is brought to you for free by Swarthmore College Libraries' Works. It has been accepted for inclusion in Senior Theses, Projects, and Awards by an authorized administrator of Works. For more information, please contact myworks@swarthmore.edu.
Application of Computer Vision to Project Sewing Patterns onto Fabric

Michelle Zhuang
Advised by Professor Matt Zucker

Abstract
This senior design project is the application of computer vision to project sewing patterns onto fabric. Utilizing key concepts of computer vision such as homographies, affine transformations, keypoint detection, and feature mapping a final demonstration script has been implemented to project patterns onto fabric in real time. Keypoints are detected from real time camera feed of the fabric’s movement to establish the affine transformation between the two. The projected pattern can then be warped to fit the fabric’s new position. This project aims to be a solution better than the current system, which requires tracing and then cutting patterns onto fabric, by allowing patterns to directly be projected onto the fabric, shifting as necessary.
Table of Contents

Abstract..............................................................................................................................................................1
Introduction..........................................................................................................................................................3
  1. Objective..................................................................................................................................................3
  2. Background..............................................................................................................................................3
Methods...............................................................................................................................................................6
  1. Physical System........................................................................................................................................6
  2a. Projection System.....................................................................................................................................6
  2b. Tracking System.......................................................................................................................................7
  3. Final Demonstration – Combining the Three Systems...............................................................................9
Results.................................................................................................................................................................11
  1. Homography Correctness.......................................................................................................................11
  2. Real Time Demonstration......................................................................................................................12
Discussion........................................................................................................................................................13
  1. Challenges...............................................................................................................................................13
  2. Future Work..............................................................................................................................................14
Conclusions.......................................................................................................................................................14
Appendix............................................................................................................................................................14
  Appendix A: autoCameraHomography.py.................................................................................................14
  Appendix B: establishHomographies.py..................................................................................................16
  Appendix C: projectMapping.py...............................................................................................................16
  Appendix D: maskFeatureAndMap.py.......................................................................................................17
  Appendix E: getBGR.py...........................................................................................................................19
  Appendix F: finalDemo.py.......................................................................................................................20
References........................................................................................................................................................23
Introduction

1. Objective

Precision is key to many sewing projects. The standard technique when using sewing patterns is a step by step process of tracing the pattern onto the fabric and then cutting out the fabric, but this can be time consuming and there can be small errors during this process if the fabric shifts or stretches. This project aims to implement a solution better than this existing method. The projection of sewing patterns onto fabric, tracing shifts in the fabric and making adjustments accordingly can help minimize errors and allow for ease in the process of sewing projects. A successful project looks like an implementation that is able to track the fabric’s movements and adjust the projections as necessary. In addition, a successful project looks like a method that is better than the current method for cutting out sewing patterns.

2. Background

Transformations map one set to another set using some operation such as affine transformations or perspective transforms. An affine transformation is a linear transformation that preserves parallel lines. A perspective transformation, known as homography, is a linear transformation that relates two planes, preserving only straight lines. Both can be defined and used through matrix multiplication (fig. 1).

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 1. Perspective transformation mapping points $(x, y)$ to new plane $(x', y')$ [9]
Morphological operations are non-linear filters used as a form of binary image processing, transforming the shapes and structures of an image. These operations include erosion, which erodes the structure of the image, making boundaries thinner, and dilation, which dilates the structure, making boundaries thicker (fig. 3).

Feature detection extracts pieces of information about an image, such as specific points or edges, which will allow for tracking the movement of the fabric when cutting out a sewing pattern. Sparse tracking uses some features (e.g., a few pixels, for example, at the corners of an object) to describe the optical flow, or image motion. OpenCV is a programming library with computer vision tools, and has example code to demonstrate the feature matching techniques (fig. 4) found in sparse tracking. Feature matching uses high interest features of images to match
keypoints between the two (fig. 4). Corners are often high interest features because they are
sharp and spatially distinct.

Figure 4. Fast Library Approximate Nearest Neighbors (FLANN) based matcher using ORB [3]

Oriented FAST and Rotated BRIEF (ORB) is a feature detection algorithm developed by
OpenCV Labs in 2011. It uses FAST (Features from Accelerated and Segments Test) to find
keypoints by comparing each pixel to the brightness of its surrounding pixels. If more than half
of its surrounding 16 pixels are classed as being darker or brighter than the given pixel, the pixel
is detected as a keypoint (fig. 5). It also uses BRIEF (Binary Robust Independent Elementary
Feature) to convert the detected keypoints into a binary feature vector, which is used as a
descriptor for each keypoint.
Figure 5. FAST algorithm classification for a given pixel $p$ [10]

Color is digitally represented as a 3D point in the red, blue, and green (RBG) colorspace (fig. 6). In this colorspace, colors are a linear combination of red, blue, and green values, where each one of the three color channels is assigned 8 bits (e.g., color intensity ranges from 0 to 255). The color channels are also correlated to the amount of light emitted.

Figure 6. RGB colorspace [8]

Methods

1. Physical System

The physical system consists of three components: a camera, a projector, and a tabletop (fig. 7). All three components must be fixed to establish homographies that relate the various planes and minimize errors for these established homographies. Each component must be fixed to their position, but should also preserve the surrounding space at the conclusion of this project, when the setup will be taken down. A wooden box to hold the projector was constructed in the
machine shop to wedge the projector between the surrounding cabinet and wall, allowing the projector to project down onto the tabletop below. The camera is taped to the same cabinet, allowing the camera to capture what the projector is projecting onto the tabletop below. A grid is fixed to the tabletop to easily measure distances.

![Image](image.png)

**Figure 7.** Physical system setup of camera, projector, and tabletop

### 2a. Projection System

The purpose of the projection system is to establish homographies to relate the three planes and perform the respective transformations that relate these planes (fig. 8). The camera frame from projector frame homography \( H_{c \ from \ p} \) is automatically calculated in `autoCameraHomography.py` (Appendix A). To establish this homography, a list of known coordinates in the projector’s frame \( (q_p) \) are projected one-by-one onto the tabletop as small circles. The mass center of each projected circle is calculated to find the corresponding coordinate in the camera’s frame \( (q_c) \). OpenCV’s `findHomography` method uses these two lists of coordinates to calculate \( H_{c \ from \ p} \). The camera frame from tabletop frame homography \( (H_{c \ from \ t}) \) is manually calculated in `establishHomography.py` (Appendix B). To establish this homography, a list of coordinates in the tabletop’s frame \( (q_t) \) are manually measured in inches from the tabletop. Demo code to determine the pixel locations of each point is used to find the corresponding coordinate in the camera’s frame. Similar to the approach before, OpenCV’s
findHomography method uses these two lists of coordinates to calculate $H_{c\text{ from } t}$. The final homography, the tabletop frame from projector frame homography ($H_{t\text{ from } p}$), is calculated through matrix multiplication of the already established homographies.

![Diagram of homographies](image)

**Figure 8.** Diagram of how the establish homographies relate the three frames

### 2b. Tracking System

The purpose of the tracking system is to use detected keypoints between two images and use feature mapping to establish the affine transformation between them. Keypoint detection and feature mapping can be found in the `maskFeatureAndMap.py` script (Appendix D). To detect keypoints within images, morphological operators were performed on the original images (fig. 9) to create a mask (fig. 11b). First, the absolute difference between the image and the paper's color, or BGR (note that the RGB colorspace is read in a BGR ordering instead), is found (fig. 10a). By fine-tuning values, a threshold is used to return a mask of where the paper is (fig. 10b). Lastly, dilation and erosion help get rid of other patterns found in the mask such that our mask primarily encapsulates the area of the paper (fig. 11b).

![Original image](image)

**Figure 9.** Original image, where paper is used in place of fabric for testing purposes
**Figure 10.** Absolute difference between image and the paper’s color (fig. 10a on left) and mask of only the paper (fig. 10b on right)

**Figure 11.** Dilation on mask (fig. 11a on left) and erosion on mask (fig. 11b on right)

This mask is passed through ORB’s keypoint detection algorithm alongside the original image so that the algorithm focuses only on features within the boundaries of the paper. Using this implementation, keypoints are detected for two images: the initial image (fig. 9) and a second image (fig. 12), after moving the paper. ORB’s feature matching algorithm then uses the keypoints and their descriptors to find matches between the two images. The top ten keypoint matches (fig. 13) are used in the `projectMapping.py` script (Appendix C) to establish the affine transformation between the two images.
3. Final Demonstration – Combining the Three Systems

The `finalDemo.py` script (Appendix F) combines the three systems discussed above. During the initial testing and implementation of each system, a hard coded BGR of the orange paper was used to create image masks. An automatic calculation of the paper’s BGR is found in `getBGR.py` script (Appendix E) to expand this system past just using hard coded values. First, an image is taken of the tabletop alone (fig. 14a) and after placing the paper onto the tabletop (fig. 14b). The absolute difference between these two images is taken to create a mask of the paper (fig. 15). The BGR of all pixels within the mask is extracted, and the BGR that is found most within the mask is returned as the BGR for the rest of the system to use.
After the BGR of the paper is calculated, the `finalDemo.py` script (Appendix F) projects an outline of a square within the boundaries of the paper – this square is used in place of a sewing pattern for testing purposes. The same masking technique from before is used to determine where the paper is located, allowing the program to get the center of the paper and project the square there.

The `finalDemo.py` script (Appendix F) then enters a constant loop such that keypoints are detected in real time, an affine transformation is created, and the projected square is warped to project onto the new position of the paper. The initial square projected onto the tabletop is in the projector’s frame. The affine transformation is established using two camera images, therefore its respective transformation matrix (`matrix_P`) is in the camera frame. Using the established homography $H_{c\_from\_p}$, the projected square image is warped to be in the camera frame. The affine transformation $matrix_P$ is performed on this image to move the square to the paper’s second position in the camera’s frame. A final transformation using the inverse of $H_{c\_from\_p}$ returns the square to the projector frame so that it can be projected onto the tabletop.
**Results**

1. **Homography Correctness**

The established homographies in the projection system were checked numerically and visually for correctness. Numerically, coordinates in their respective frames are transformed into other frames. The difference between the actual measured coordinates and the calculated coordinates through homography are found. Below (fig. 16), shows the error for the \( H_{c\_from\_p} \) and \( H_{c\_from\_t} \) homographies in pixel units. Visually, coordinates in the camera’s frame are warped by the inverse of \( H_{c\_from\_p} \) to return coordinates in the projector’s frame. These coordinates are visually compared to the original known projector frame coordinates used to automatically calculate the homography by overlapping the two (fig. 17).

![Error for camera from projector homography](image1.png)

*Figure 16. Homography error between camera and tabletop and between camera and projector*

![Error for camera from table homography](image2.png)

*Figure 17. Black circles are the known projector points and red circles are camera points warped into the projector frame*
2. **Real Time Demonstration**

The final demonstration correctly identifies the BGR of paper to establish a mask of where the paper is on the tabletop. The projected square is warped from the initial position to each position afterward (fig. 18). The initial position and its projected square can be transformed to mimic the second position of the paper. Overlaying this with the second position and its projected square shows that the square was properly transformed (fig. 19). Additional real time projection of the square occurs in this same way (fig. 20).

![Figure 18](image1.png)  
**Figure 18.** Original paper’s position and projected square (fig. 18a on left) and paper’s second position with warped projected square (fig. 18b on right)

![Figure 19](image2.png)  
**Figure 19.** Two images overlaid to check that the warped transformation aligned correctly
Discussion

1. Challenges

The physical system must be rigid to ensure that the established homographies to relate each frame remain correct throughout each trial. Since the system is in a shared room, attached to a set of cabinets, shifts in the camera’s position have slowly occurred over time. This motivated a shift in implementation from manually to automatically calculating the camera from projector homography. The webcam currently used has issues focusing sometimes due to its light sensitivity. This poses a challenge because blurred images captured by the camera inhibits keypoint features to properly be detected. Without matching keypoints, the system is unstable to establish an affine transformation between the position of the paper. A five second delay is currently in place if keypoints cannot be detected to allow for the camera to focus again. A better camera might avoid this issue in the future.
2. Future Work

In real time, the frame rate is slow, but this is not due to a correctness issue. Future work could look into using a faster language than Python or a faster system for tracking than OpenCV. Thus far, testing has been conducted on various colors of paper with “x’s” drawn to mimic patterned fabric. Patterns are easily recognized features for the keypoint detection to register and use for feature mapping. This is vital to how the current program tracks the movement of the object and establishes the affine transformation between positions. In future work, this system could be expanded to work on both patterns and solid fabrics. Additionally, some fabrics are stretchier than others (e.g., canvas vs silky rayon). In future work, this system might need to consider the shear of fabrics, rather than just \((x, y)\) coordinates of the top keypoint matches.

The initial evaluation of this project was to create an alternative better than the current system: tracing and cutting fabric. Due to time constraints, most of this project consisted of design and implementation, but in future work, more time would be used to assess the accuracy and timing of this project for cutting fabric.

Conclusions

Utilizing key concepts of computer vision such as homographies, affine transformations, keypoint detection, and feature mapping a final demonstration script has been implemented to project patterns onto fabric in real time. The paper’s BGR is automatically detected by the program, using this color to create masks of where the paper is on the tabletop. Keypoints are detected from real time camera feed between the paper’s movement to establish the affine transformation between the two. The projected pattern can then be warped to fit the fabric’s new position. Future work to compare this project’s solution to the current standard of tracing and then cutting sewing patterns onto fabric will be useful in evaluating the goodness of this project.
Appendix

Appendix A: autoCameraHomography.py

```python
import numpy as np
import cv2

import homographyErrors
import homographyVisualDebug

def main():
    # project white background
    display = np.full((800, 1400, 3), 0, dtype=np.uint8)

    q_p = np.genfromtxt('data_points/projector_points_mar23.txt')
    cam = cv2.VideoCapture(0)

    cv2.namedWindow("Projected Image")
    cv2.moveWindow("Projected Image", 1792, 0)
    cv2.namedWindow("Camera Image")

    q_c = []

    initial_img = takeImage(cam, display, 0)
    for i in range(len(q_p)):
        background = display.copy()
        dot = cv2.circle(background, (int(q_p[i][0]), int(q_p[i][1])), 20, (255,255,255), -1)
        cameraImg = takeImage(cam, dot, i+1)
        x, y = getMassCenter(initial_img, cameraImg)
        q_c.append([x,y])

    q_c = np.array(q_c)
    H_c_from_p, mask = cv2.findHomography(q_p, q_c)

    print("\nthe homography from Projector to Camera: ", H_c_from_p)

    homographyErrors.getCameraError(q_p, q_c, H_c_from_p)
    homographyVisualDebug.visualError(q_p, q_c, H_c_from_p)

    np.savetxt('tools/H_c_from_p_mar23.txt', H_c_from_p, fmt='%.5f')
    np.savetxt('data_points/camera_points_debug_mar23.txt', q_c, fmt='%.5f')

    cv2.destroyAllWindows()

def getMassCenter(img1, img2):
    diff1 = cv2.absdiff(img1, img2)

    cv2.imshow('Camera Image', diff1)
    print('hit space again')
    while cv2.waitKey(5) != ord(' '): pass
```

diff2 = cv2.cvtColor(diff1, cv2.COLOR_BGR2GRAY)
__, diff3 = cv2.threshold(diff2, 16, 255, cv2.THRESH_BINARY)

cv2.imshow('Camera Image', diff3)
print('hit space again')
while cv2.waitKey(5) != ord(' '): pass

contours, _ = cv2.findContours(diff3, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# get biggest contour
_c = max(contours, key = cv2.contourArea)
moment = cv2.moments(_c)
mass_center = (moment['m10'] / moment['m00']), moment['m01'] / (moment['m00']))

x, y = mass_center

display = img2.copy()
cv2.circle(display, (int(x), int(y)), 5, (255, 0, 255), -1, cv2.LINE_AA)
cv2.imshow('Camera Image', display)
print('hit space again')
while cv2.waitKey(5) != ord(' '): pass

return mass_center

if __name__ == '__main__':
    main()

Appendix B: establishHomographies.py

def homographies():
    
    q_t = np.genfromtxt('data_points/table_points_mar24.txt')
    q_c = np.genfromtxt('data_points/camera_points_mar24.txt')

    print('got q_c:', q_c)
    print('got q_t:', q_t)

    H_c_from_p = np.genfromtxt('tools/H_c_from_p_mar23.txt') # using autocalibrated H_c_from_p
    print("the homography from Projector to Camera: ", H_c_from_p)

    H_c_from_t, mask = cv2.findHomography(q_t, q_c)
    print("the homography from Camera to Tabletop: ", H_c_from_t)

    H_t_from_c = np.linalg.inv(H_c_from_t)

    H_t_from_p = H_t_from_c @ H_c_from_p

    np.savetxt('tools/H_c_from_t_mar24.txt', H_c_from_t, fmt="%f")
Appendix C: projectMapping.py

```python
def getMapping(kpl_matches, kp2_matches, H_c_from_t):
    kpl_tableCoords = cv2.perspectiveTransform(kpl_matches.reshape(-1,1,2),
                                              np.linalg.inv(H_c_from_t))
    kp2_tableCoords = cv2.perspectiveTransform(kp2_matches.reshape(-1,1,2),
                                              np.linalg.inv(H_c_from_t))

    if (type(kpl_tableCoords)!=None and type(kp2_tableCoords)!=None):
        if (len(kpl_tableCoords) == len(kp2_tableCoords)):
            # affine transformation in table coordinates
            matrix_A, inliers = cv2.estimateAffine2D(kpl_tableCoords, kp2_tableCoords)
            matrix_M = np.vstack((matrix_A, [[0, 0, 1]]))
            matrix_P = H_c_from_t @ matrix_M @ np.linalg.inv(H_c_from_t)
            return True, matrix_P

        return False, None

def getError(kpl_tableCoords, kp2_tableCoords, matrix_M):
    input_points_reshaped = kpl_tableCoords.reshape((-1,1,2))
    output_points_reshaped = cv2.perspectiveTransform(input_points_reshaped, matrix_M)
    output_points = output_points_reshaped.reshape(-1,2)
    error = kp2_tableCoords.reshape(-1,2) - output_points
    print("\nAffine Transformation Error: ", error)
```

Appendix D: maskFeatureAndMap.py

```python
def createMask(img, BGR, debug_mode):
    # find all the BGR color pixels
    diff = np.abs(img - BGR).max(axis=2)
    diff = diff.astype(np.uint8)
    diff2 = np.where(diff < 50, np.uint8(255), np.uint8(0))
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (21, 21))
    diff3 = cv2.dilate(diff2, kernel, iterations = 1)
    diff4 = cv2.erode(diff3, kernel, iterations = 1)

    if debug_mode:
        show(diff)
        show(diff2)
        print('after dilation')
        show(diff3)
        show(diff4)

    return diff4
```
def getMatches(img1, img2, BGR, debug_mode):
    if debug_mode:
        show(img1)
        show(img2)

    # return morphological operated image to use as mask
    mask1 = createMask(img1, BGR, debug_mode)
    mask2 = createMask(img2, BGR, debug_mode)

    orb = cv2.ORB_create()

    # find the keypoints and descriptors with ORB
    kp1, des1 = orb.detectAndCompute(img1, mask1)
    img1 = cv2.drawKeypoints(img1, kp1, None, color=(0, 255, 0), flags=0)
    kp2, des2 = orb.detectAndCompute(img2, mask2)
    img2 = cv2.drawKeypoints(img2, kp2, None, color=(0, 255, 0), flags=0)

    if debug_mode:
        show(img1)
        show(img2)

    kp1_coords = cv2.KeyPoint.convert(kp1)
    kp2_coords = cv2.KeyPoint.convert(kp2)

    if (des1 is not None and des2 is not None):
        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

        # Match descriptors.
        matches = bf.match(des1, des2)

        # Sort them in the order of their distance.
        matches = sorted(matches, key=lambda x: x.distance)

        # Draw first 10 matches.
        img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

        if debug_mode:
            show(img3)

        # save matches
        top_matches_kp1 = []
        top_matches_kp2 = []
        for match in matches[:10]:
            p1 = kp1[match.queryIdx].pt
            p2 = kp2[match.trainIdx].pt
Appendix E: getBGR.py

```python
def getBGR(cam):
    cam = cv2.VideoCapture(0)
    cv2.namedWindow("BackgroundImg")
    cv2.moveWindow("BackgroundImg", 1792, 1000)
    background = np.full((800, 1400, 3), 255, dtype=np.uint8)
    cv2.imshow("BackgroundImg", background)

    before = takeImage(cam, "background.png")
    after = takeImage(cam, "after.png")

    diff1 = cv2.absdiff(before, after)
    diff2 = cv2.cvtColor(diff1, cv2.COLOR_BGR2GRAY)
    diff3 = diff2.astype(np.uint8)
    mask = np.where(diff3 > 33, np.uint8(255), np.uint8(0))
    kernel = np.ones((7,7), np.uint8)
    erosion = cv2.erode(mask, kernel, iterations = 1)
    bgr_colors = after[mask.astype(bool)] # n by 3 array
    bgr_colors = bgr_colors >> 2
    bgr_colors = bgr_colors << 2
    unique_bgr, counts = np.unique(bgr_colors, axis=0, return_counts=True)
    BGR = unique_bgr[counts.argmax()] # BGR of paper
    std_dev = np.std(unique_bgr, axis = 0)
    print("Got the BGR of paper!")
    print("R: ", BGR[2])
    print("G: ", BGR[1])
    print("B: ", BGR[0])

    print("The standard deviation across channels are:")
    print(std_dev)
    cv2.destroyWindow("debugImg")
    return BGR, std_dev
```

```python
def main():
    q_p = np.genfromtxt('data_points/projector_points_mar23.txt') # input points to project
```
```python
q_t = np.genfromtxt('data_points/table_points_mar24.txt')
q_c = np.genfromtxt('data_points/camera_points_debug_mar23.txt')

H_c_from_p = np.genfromtxt('tools/H_c_from_p_mar23.txt')
H_c_from_t = np.genfromtxt('tools/H_c_from_t_mar24.txt')

input_points_reshaped = q_t.reshape((-1,1,2))

### Camera and Projector Homography ###
output_points_reshaped = cv2.perspectiveTransform(input_points_reshaped, H_c_from_p)
output_points = output_points_reshaped.reshape(-1, 2)
error = q_c - output_points
print("Error for camera from projector homography: ", error)

input_points_reshaped = q_t.reshape((-1,1,2))
output_points_reshaped = cv2.perspectiveTransform(input_points_reshaped, H_c_from_t)
output_points = output_points_reshaped.reshape(-1, 2)
error = q_c - output_points
print("Error for camera from table homography: ", error)

def getCameraError(q_p, q_c, H_c_from_p):
    input_points_reshaped = q_p.reshape((-1,1,2))
    output_points_reshaped = cv2.perspectiveTransform(input_points_reshaped, H_c_from_p)
    output_points = output_points_reshaped.reshape(-1, 2)
    error = q_c - output_points
    print("\nError for camera homography: ", error)

Appendix F: finalDemo.py

import numpy as np
import cv2
import sys

# from tools import cameraCapture
from tools_featureMapping import maskFeatureAndMap
from tools_featureMapping import projectMapping
from tools_homography import establishHomography
from tools import getBGR

H_t_from_p, H_c_from_p, H_c_from_t = establishHomography.homographies()

def takeImage(cam, background, img_name):
    h, w = background.shape[:2]
    while True:
        cv2.imshow("BackgroundImg", background)
        ret, frame = cam.read()
        if not ret:
            print("failed to grab frame")
```
break

cv2.imshow("CameraImg", frame)
k = cv2.waitKey(1)
if k%256 == 32:  # SPACE pressed
    print("captured image: ", img_name)
    break

cv2.imwrite(img_name, frame)
return frame

def show(img):
    while True:
        cv2.namedWindow("ProjectedImg", cv2.WINDOW_NORMAL)
        cv2.imshow("ProjectedImg", img)

        k = cv2.waitKey(1)
        if k%256 == 27:  # ESC pressed
            break

def createSquare(img1, background, BGR, debug_mode):
    img1_copy = img1.copy()
    mask1 = maskFeatureAndMap.createMask(img1_copy, BGR, debug_mode)
    ret, thresh = cv2.threshold(mask1, 127, 255, 0)
    contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(img1_copy, contours, -1, (0,255,0), 3)
    cnt = max(contours, key = cv2.contourArea)

    x, y = getCenter(cnt)  # in camera coordinates
    cv2.circle(img1_copy, (int(x), int(y)), 10, (0,0,0), -1)
    q_c = np.float32([[x,y]])
    q_p = cv2.perspectiveTransform(q_c.reshape((-1,1,2)),
        np.linalg.inv(H_c_from_p)).reshape(-1, 2)
    x, y = q_p[0][0], q_p[0][1]
    start_point, end_point = (x-50, y+50), (x+50, y-50)
    h, w = background.shape[:2]
    cv2.rectangle(background, (int(start_point[0]), int(start_point[1])),
        (int(end_point[0]), int(end_point[1])), (0,0,0), 7)

    h, w = background.shape[:2]
    cv2.imwrite("tools/square.png", background)

return background

def getCenter(cnt):
    leftmost = tuple(cnt[:,0].argmin())
    rightmost = tuple(cnt[:,0].argmax())
topmost = tuple(cnt[cnt[:, :, 1].argmin()][0])
bottommost = tuple(cnt[cnt[:, :, 1].argmax()][0])
x = (leftmost[0] + rightmost[0]) / 2
y = (topmost[1] + bottommost[1]) / 2
return x, y

def overlayDebug(img1_withSquare, img2_withSquare, matrix_P):
    h, w = img1_withSquare.shape[:2]
    img1_debug = cv2.warpPerspective(img1_withSquare, matrix_P, (w, h))
    debug = cv2.addWeighted(img1_debug, 0.5, img2_withSquare, 0.5, 0)

def main():
    debug_mode = len(sys.argv) > 1 and sys.argv[1] == 'debug'
    cam = cv2.VideoCapture(0)
    cv2.namedWindow("BackgroundImg")
    cv2.moveWindow("BackgroundImg", 1792, 1000)
    background = np.full((800, 1400, 3), 255, dtype=np.uint8)

    if debug_mode:
        background_img = cv2.imread('background.png')
        src = cv2.imread('after.png')
    else:
        background_img = takeImage(cam, background, "background.png")
        src = takeImage(cam, background, "after.png") # press SPACE to take image

    # Get the BGR of paper
    bgr, std_dev = getBGR.returnBGR(background_img, src)

    square1 = createSquare(src, background.copy(), bgr, debug_mode)
    square = square1.copy()

    if debug_mode:
        img1_withSquare = cv2.imread('initial_square.png')
    else:
        img1_withSquare = takeImage(cam, square1, "initial_square.png")

    while(True):
        cv2.imshow("BackgroundImg", square)

        if debug_mode:
            frame = cv2.imread('latest_frame.png')
        else:
            ret, frame = cam.read()
            cv2.imshow("CameraImg", frame)
            if not ret:
                print("failed to grab frame")
                break
cv2.imwrite('latest_frame.png', frame)

error, kp1_matches, kp2_matches = maskFeatureAndMap.getMatches(src, frame, bgr, debug_mode)
if error:
    error2, matrix_P = projectMapping.getMapping(kpi_matches, kp2_matches, H_c_from_t)
    if error2:
        h, w = square1.shape[:2]
        square1_tocopy = cv2.warpPerspective(square1.copy(), H_c_from_p, (w, h))
        square2 = cv2.warpPerspective(square1_tocopy, matrix_P, (w, h))
        square2 = cv2.warpPerspective(square2, np.linalg.inv(H_c_from_p), (w, h))
        square = square2.copy()

k = cv2.waitKey(1)
if k % 256 == 27:  # ESC pressed
    break

cam.release()
cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
References


