University of Tennessee at Chattanooga

UTC Scholar

Honors Theses

Student Research, Creative Works, and Publications

5-2020

Augmented Reality in a Dynamic Drone Based Environment

Christopher Davis
University of Tennessee at Chattanooga, nzg321@mocs.utc.edu

Follow this and additional works at: https://scholar.utc.edu/honors-theses

Part of the Aeronautical Vehicles Commons, and the Computer Engineering Commons

Recommended Citation

Davis, Christopher, "Augmented Reality in a Dynamic Drone Based Environment" (2020). *Honors Theses.*

This Theses is brought to you for free and open access by the Student Research, Creative Works, and Publications at UTC Scholar. It has been accepted for inclusion in Honors Theses by an authorized administrator of UTC Scholar. For more information, please contact scholar@utc.edu.

Augmented Reality in a Dynamic Drone Based Environment

Chris Davis

College of Engineering and Computer Science

University of Tennessee at Chattanooga

Supervisor

Dr. Yu Liang

Undergraduate Thesis

Bachelor of Science in Computer Science

Examination Date: March 27, 2020

Abstract

Augmented Reality (AR) and Unmanned Ariel Vehicles (UAVs) are fast advancing technologies, and this research seeks to combine them to offer an effective, user friendly approach for monitoring infrastructure. Drones provide a means to easily access otherwise difficult to reach locations and visualize useful information with Augmented Reality. A UAV employs a wide-angle view and, when paired with AR, this will enable the user to better complete their task by effortlessly providing the critical information they need in the most intuitive way possible. This research is particularly applicable for civil applications such as construction and monitoring of difficult to access locations. The background for these technologies and applications will be discussed in the next section, but this research is unique in its combined and interactive application of AR and drone technology. Many existing techniques which are discussed in this paper already exist for general applications of AR. Using a drone feed compared to a relatively fixed camera creates many barriers to the AR process. This research explores how existing techniques perform under these conditions, such as fast moving cameras and complex environments that drones often face. This research also explores potential methods for improving accuracy for AR objects created on a Drone feed. Existing technology such as object tracking and image filtering are used to improve accuracy. Other simple mathematical methods are used, such as Kalman Filtering and data smoothing algorithms, to improve the appearance of the AR object in the frame.

Contents

1 Introduction **1**

1.1 Background 1

2 Methods **4**

2.1 Augmented Reality Implementation 4

2.2 Object Tracking and Frame Masking 6

2.3 Output Smoothing 7

3 Results **9**

3.1 Improvements to Traditional AR Methods 9

3.2 Program Performance 9

3.3 Program Output 10

4 Discussion **17**

4.1 The Future of Drone Based AR 17

4.2 Future Improvements 17

5 Conclusions **19**

References **20**

6 Appendix **21**

6.1 Full Python Source 21

Chapter 1

Introduction

1.1 Background

AR or Augmented Reality is the projection of a virtual object into a live or recorded video or image. This already has many widespread applications, such as Pokémon Go or Snapchat filters, Google Translate, Google Maps, Amazon's AR View and many more. The basis for this technology is key points. Key points are points calculated from an image with an associated Matrix of values that can identify it in the frame. There are many different algorithms that can generate these key points, which will be discussed in detail in the Methods section of this paper, but they all calculate these values based on the image's intensity at a point, or the rate at which the brightness changes between a pixel and surrounding pixels. This is usually measured from a greyscale image. For any AR application there are two important components to take into account, the model and the frame. The frame is simply the image that an AR entity will be projected onto, and the model is the object in the frame the AR entity needs to be anchored to. Key points need to be generated for both the model and the frame, and then they are matched, to make sure the objects location and orientation are correct. The resulting key points are matched, and then they are sorted by confidence level. After this step, all that remains is to compute homography of the surface, and then render the object. Homography is a two-dimensional plane calculated from the collection of key points. Rendering the object is the easiest step, since, from the key points, a plane has already been generated, the object just needs to be

drawn orthogonally on that plane. While there are different ways to implement these steps, all AR works in this way. For the purposes of this research, a drone refers to a UAV, or unmanned aerial vehicle, which is a battery powered, propeller driven remote control device that has a mounted video camera. That camera can stream to a mobile phone or laptop. The object of this research is to accurately project an object onto the video feed, and make it behave as you would expect an object actually being recorded, with a constant orientation and position relative to other objects in the frame. Potential challenges to this research are abundant. The most widespread application for Augmented reality is on a mobile phone. The camera on a phone is relatively fixed and moves/rotates at relatively slow speeds when compared with a camera attached to a drone. The UAV can rotate 360 degrees in a matter of seconds, and it can easily travel at a speed of 50 miles per hour. Beyond this, drones often fly in highly dynamic environments, such as construction sights, or over broad areas such as agricultural land. These environments are often feature saturated, making key point detection computationally taxing, or feature poor, making key point matching difficult. All of these potential issues will be addressed in this research. Applications of this research are abundant, specifically in the area of construction and civil engineering. This technology would enable virtual object overlays in a drone stream. This could be useful for viewing infrastructure that isn't visible on inspection, such as underground or obstructed pipes or wires. Drones are already used for applications such as bridge inspection and construction. For bridge inspection, important areas could be highlighted so the technician has an easier job finding it. In construction, drones are used to monitor progress and develop plans, and both processes could be enhanced by the addition of augmented reality. Another notable application is entertainment. Augmented reality is extremely popular on mobile platforms for entertainment, and recreational drones could by extending their functionality to support augmented reality. Because of the high demand and applications there are for these technologies, a lot of research has been done involving drones and AR. These areas still have many ways they can be improved, and research continues to be done. In a survey the most beneficial applications of civil applications of drone technology, it was observed that drone research has been growing ex-

ponentially in recent years. It also notes the applications are widespread across environmental monitoring, land surveying, and infrastructure monitoring [1].Research involving AR has also been extensive in recent years. This particular study addresses the challenges of outdoor AR and how a user would use it. It describes multiple techniques for displaying large scale AR data in an unpredictable outdoor environment [2]. Other research has also been targeted at civil applications using both UAV's and Augmented Reality, but very little has been done with both technologies to create an interactive environment [3]. This project seeks to extend much of this previous research into a combined user experience enabled by AR and drone technology. With this advanced technology, there exist many challenges that this research seeks to overcome. A major issue with rendering virtual objects on a real 3D background is anchoring the object to a fixed point in the image and rotating/ scaling/ translating it correctly as the observation point changes. This is especially problematic when the observation point (a UAV) is moving very quickly. A drone may also create a unpredictable user experience because of it's quick motion and generally low precision sensors.

Chapter 2

Methods

2.1 Augmented Reality Implementation

For this research, python was used to load the drone videos, and python OpenCV (Computer Vision) was used for augmented reality. As discussed in the introduction, the basis of augmented reality is key points generated for an image or frame. This research employed the ORB (Orient FAST Rotate BRIEF) algorithm from OpenCV to create key points. According to OpenCV's documentation, ORB is a computationally efficient substitute for the SIFT or SURF key point detection algorithms. This computational efficiency makes it a clear candidate for this application, with drone video's generally having high resolution and dynamic scenes. Another important factor is SIFT and SURF are patented algorithms, while ORB is not patented and is free to use. ORB is a modified combination of FAST key point detection and the BRIEF descriptor algorithms. FAST (Features from Accelerated Segment Test) is a machine learning algorithm that detects corners (or features) in an image. This is accomplished by comparing the intensity of a center pixel with a circle of pixels around it. A machine learning algorithm is used to detect interest points and an intensity threshold value relative to the rest of the frame. A pixel determined from the interest point algorithm is considered a corner when there is a contiguous set of pixels around it that are all darker or lighter than it, illustrated in the figure 2.1. Once the features are detected using FAST, BRIEF (Binary Robust Independent Elementary Features) is used to calculate descriptors for the features. This algorithm generates a vector of binary features that is later

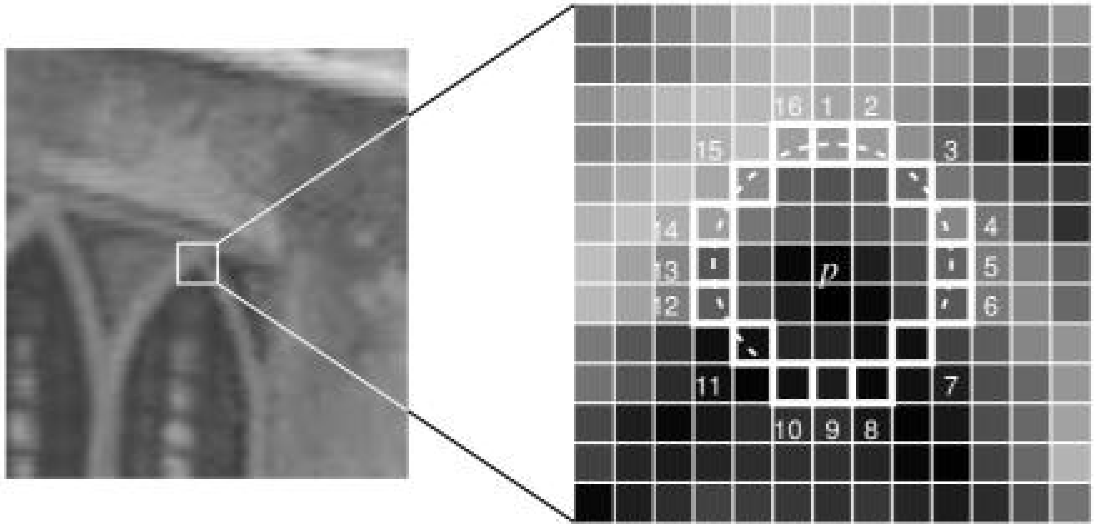

Figure 2.1 FAST Feature Detection

used for matching. An example of the ORB key point creation is shown in the Results section Figure 3.1. There are two images that the above process is applied to, the model and the current frame. The model can be instantiated many ways, but it is supposed to be the actual object in the frame where you want an AR object to be anchored. In this application, the frame is the current frame as it is recieved from the drone from the drone, and the object the user selects in an initial frame is defined as the model. With features generated from both images, OpenCV provides multiple methods for matching features between them. Because of the limitation of patented algorithms, the Brute Force matcher was chosen. The Brute Force matcher simply compares the values of the features and the minimum Euclidian distance between two features (generated by BRIEF) is selected as a match. An example of the Brute Force Matcher in application is shown in Figure 3.2 and 3.3. For this research, the best 20 matches were selected as features used to create homography and project the image onto the frame. Homagraphy is generated using openCV's functionality. It is a way of relating the transformation between two planes, and in this case the frame from the model object to where it is in the frame. It is represented as a normalized 3X3 matrix. As illustrated in figure 2.2, a plane is generated by drawing lines between key points.

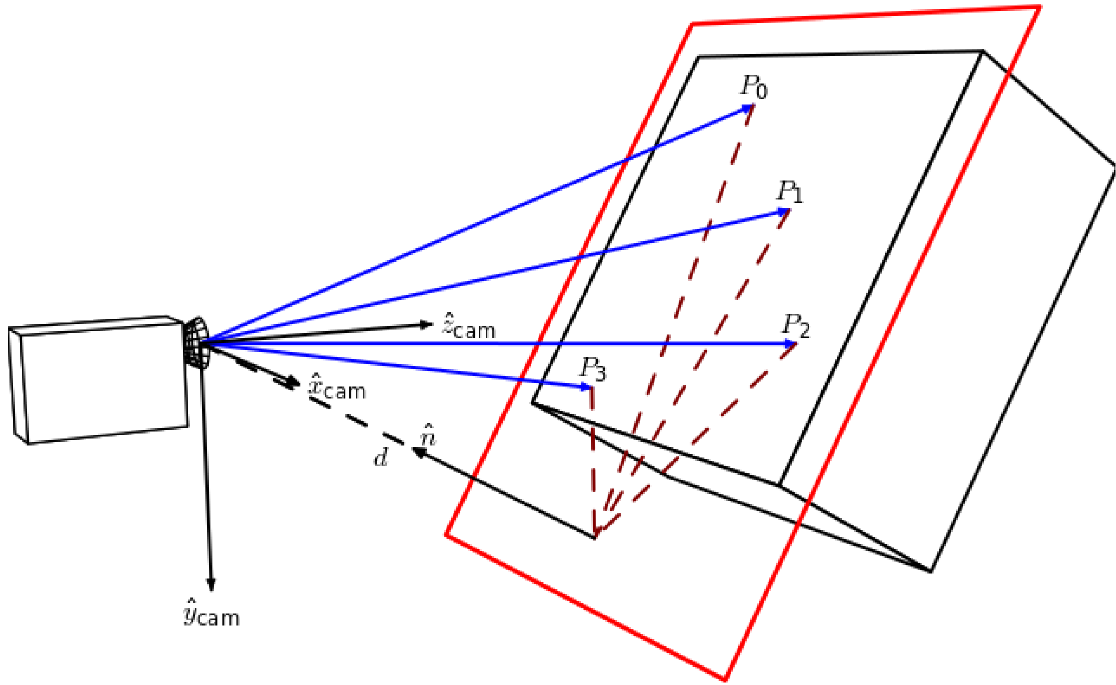

Figure 2.2 Homography Theory

2.2 Object Tracking and Frame Masking

Another important method employed in this research is object tracking. Python OpenCV provides object tracking functionality through python as well. Many algorithms are available for object tracking, but this research was developed using the KCF and the CSRT Trackers. Both trackers are based on Correlation filters. KCF was chosen for its high accuracy compared to its relatively low computational cost. The only notable downside to KCF is it fails if the object it is tracking ever becomes fully occluded, it will not be able to discover it again. CSRT performed better with occlusion and overall, but at a slightly lower speed. The reason Object tracking is relevant to AR is how it can improve key point detection and matching. The CSRT tracker is demonstrated on a drone video in figures 3.5 and 3.6. Using object tracking, a mask can be applied to each frame of the drone feed to reduce it to a much smaller region of interest (ROI). A region of interest can be created by the user, figure 3.4, and then a mask is applied to the frame, figure 3.7, and then the ROI is updated by the tracker each frame, along with the mask. This means that the feature creation and matching algorithms only need to be applied to a small subset of the image, making it significantly more accurate and efficient. This is illustrated in the arrays in figure **??**, since most of the frame after the mask

```
>>> mask
array([[  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0., 229.,  48.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.]])
>>> frame = np.random.randint(0,255,(10,10))
>>> frame
array([[185,  56,  95, 129, 132, 228,  82, 215,  68, 191],
       [ 65,  77, 188, 200, 165, 142,  36,  42,   4,   4],
       [ 91,   1, 161, 113,  67,  23, 217, 174,  32, 154],
       [254,  60,  63, 223, 129, 206, 172, 220, 214,  27],
       [170, 240,  59,  93, 176,  27,  92, 209,  98,  96],
       [175,  99, 174,  50,  96, 230, 105, 123,  49,  84],
       [ 27, 217, 125,  70,   6, 123,  50, 199, 140,  83],
       [249,  53, 175,  97, 157,  56,  84, 185, 111,  73],
       [ 18,  29, 143, 205,  64,  26,  13, 103,  79, 222],
       [213,  24, 193,  40, 118, 127,  79,  83, 152,  54]])
>>> mask = np.zeros(frame.shape)
>>> mask[7:9,6:8] = frame[7:9,6:8]
>>> mask
array([[  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,  84., 185.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,  13., 103.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.]])
```

Figure 2.3 Python Mask Computation

is zero, the calculations that need to be done are drastically reduced.

2.3 Output Smoothing

A few other mathematical techniques were applied to further improve performance, including averaging homography and Kalman filtering. Ideally, an AR object would have smooth motion as it traveled in the frame. Since many calculations are made every frame, and sometimes noise is introduced, a few smoothing methods were applied to improve performance relative to the user. The simplest technique applied was averaging past values for homography with the current one. Through trial and error, averaging the past five to ten frames best smoothed the object in the frame without sacrificing accuracy. Kalman filtering was also applied to the homography matrix. Kalman filtering is a signal noise processing technique that has been used for decades. It consists of noise matrices, a state matrix, and an update matrix that allow for real time signal processing to give smoother feedback than the actual measurement. When applied to the homography matrix, it helped

reduce the amount the object would jump within the frame.

Chapter 3

Results

3.1 Improvements to Traditional AR Methods

Each of the methods described above had an impact on the performance (as judged by the user) of the program. Traditional AR methods generally rely on key point matching alone to match objects between frames, but that approach was not sufficient for the dynamic environments that drones are often in. While the traditional methods worked for a fixed camera and a predetermined model, the accuracy of key point matching did not accurately project the object in the image from frame to frame when applied to a drone video feed. The addition of masking the object based on the object tracking before key point matching is what made AR work for each drone feed sample. The additional smoothing made the object seem more realistic in the frame. Many of these measures are arbitrary, so sample output is included in section 3.3, specifically in figures 3.13, 3.13, and 3.11.

3.2 Program Performance

Together, all these methods are applied together to enable Augmented Reality in drone video. This research successfully and accurately achieved the result of anchoring an AR object (in this case a simple cube) and orienting it correctly in the frame of a drone feed. With the available resources, the python script runs at a frame rate between 3 and 6 frames per second running on a 2.7 GHz processor with 16 GB of Memory. With more computational power, or some code optimization,

this number could reach 24 frames/sec, the usual frame-rate for a 4K drone stream. As far as the positional accuracy and the pose of the AR object, the code works well on surfaces with good features, but can perform poorly on surfaces with few unique points. When there are quality features, the program does a great job of keeping an objects location, and an acceptable job of estimating the change in pose over time. When dealing with occlusion of the model, the program will drop the AR object from the frame until it returns into view according to the object tracker. This program provides an easy working product that is editable for a user. All that would need to be done to implement is download or record an mp4 file from a drone's camera and point the program to it. The user can also specify an AR object to project onto the frame in the form of an object (.obj) file. Once the program is run, it prompts the user to select a subset of the image as a bounding box on which to anchor the AR object. Once the user selects this location, the program tracks the location in the frame and projects the AR object onto the frame. This research is unique in its specific combination of the techniques of object tracking, frame masking, and Kalman filtering to achieve accurate Augmented Reality in the dynamic setting of Drone videos.

3.3 Program Output

This section contains samples output from the program developed in this research applied to multiple open source drone videos.

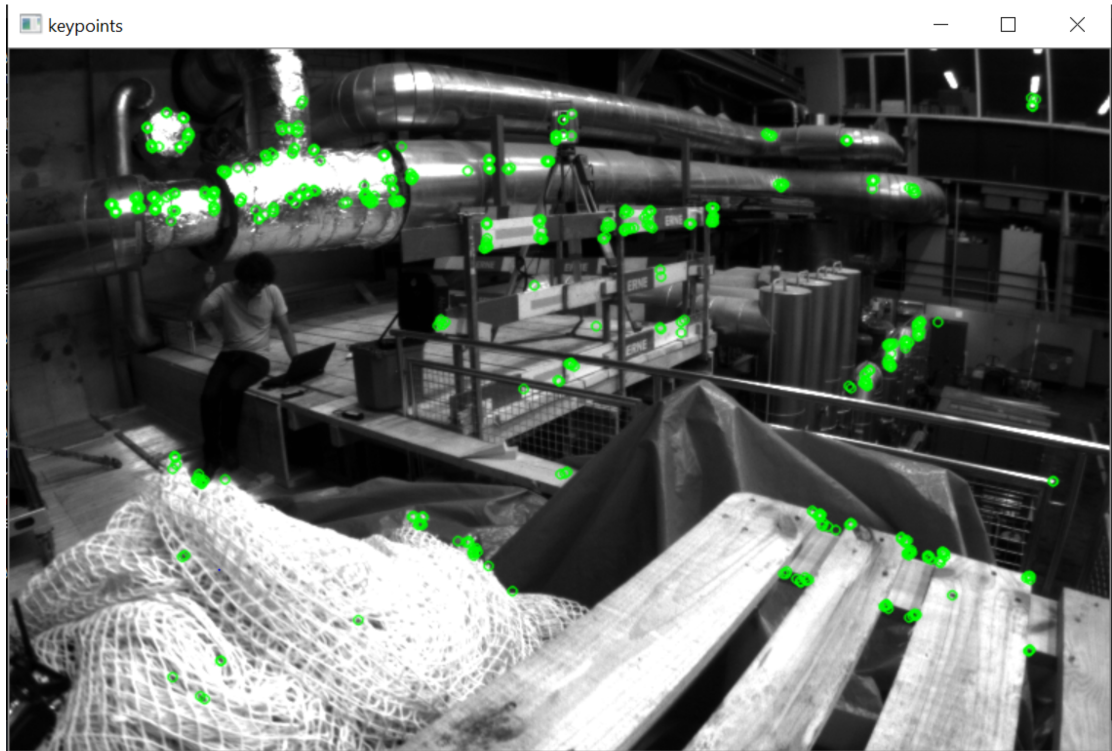

Figure 3.1 Key Point Creation with ORB

Figure 3.2 Key Point Matching with Brute Force Matcher

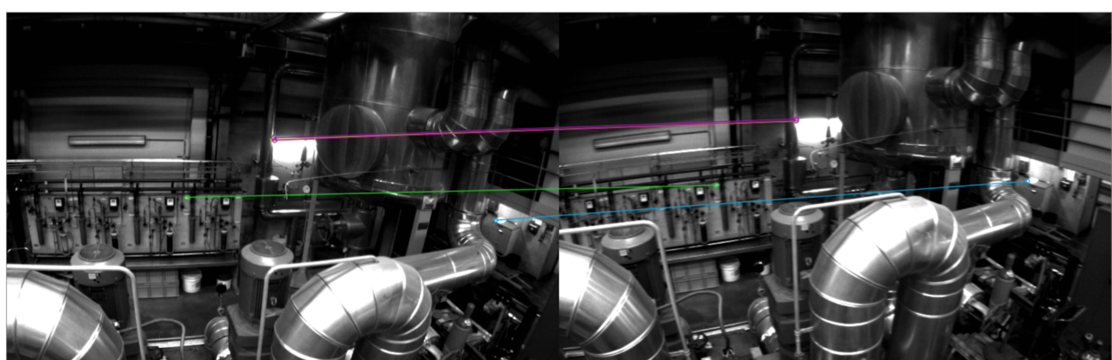

Figure 3.3 Key Point Matching with Brute Force Matcher (High Confidence Points Only)

Figure 3.4 Selecting a Region of Interest

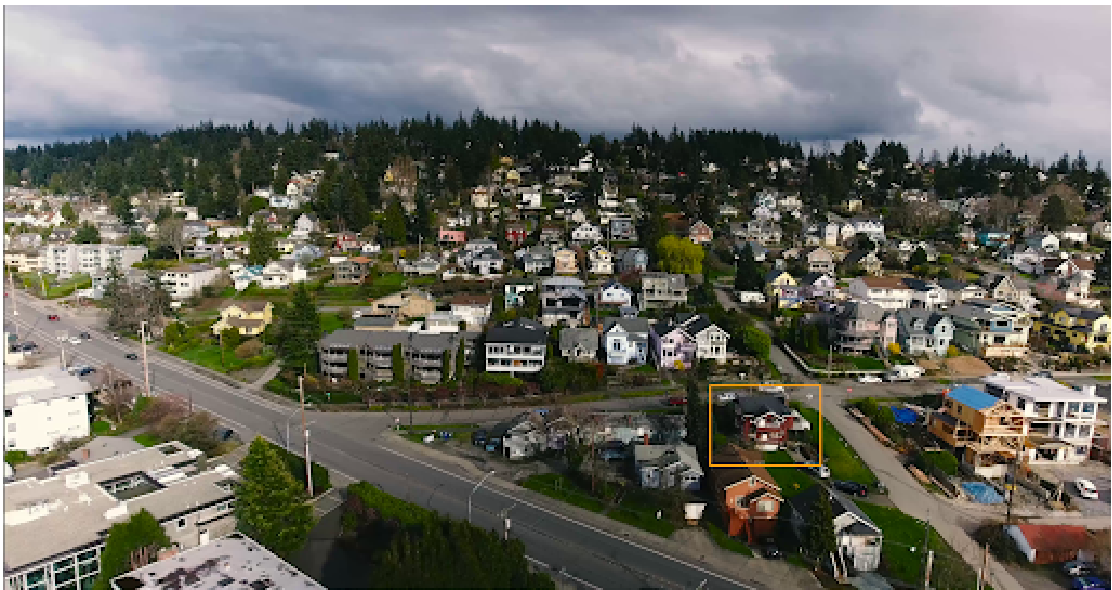

Figure 3.5 Simple Object Tracking at Beginning of Feed

Figure 3.6 Simple Object Tracking at End of Feed

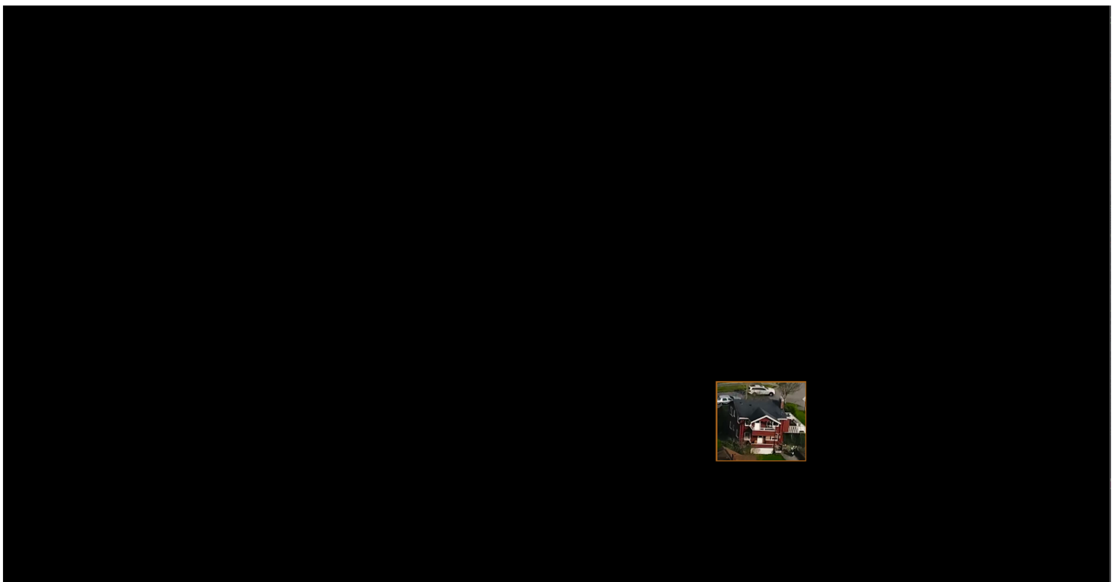

Figure 3.7 Using Object Tracking to Apply a Mask

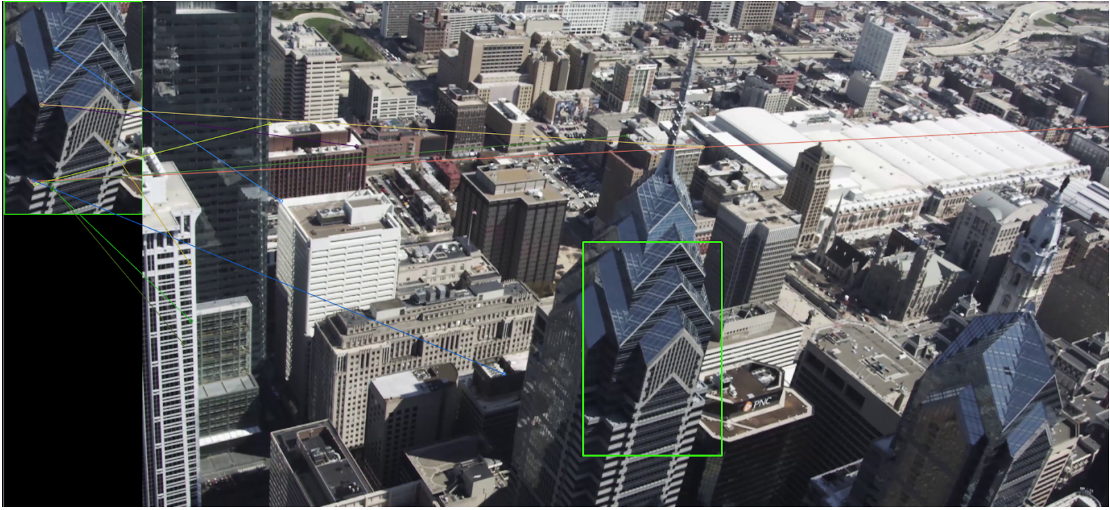

Figure 3.8 Key Point Matching without Masking

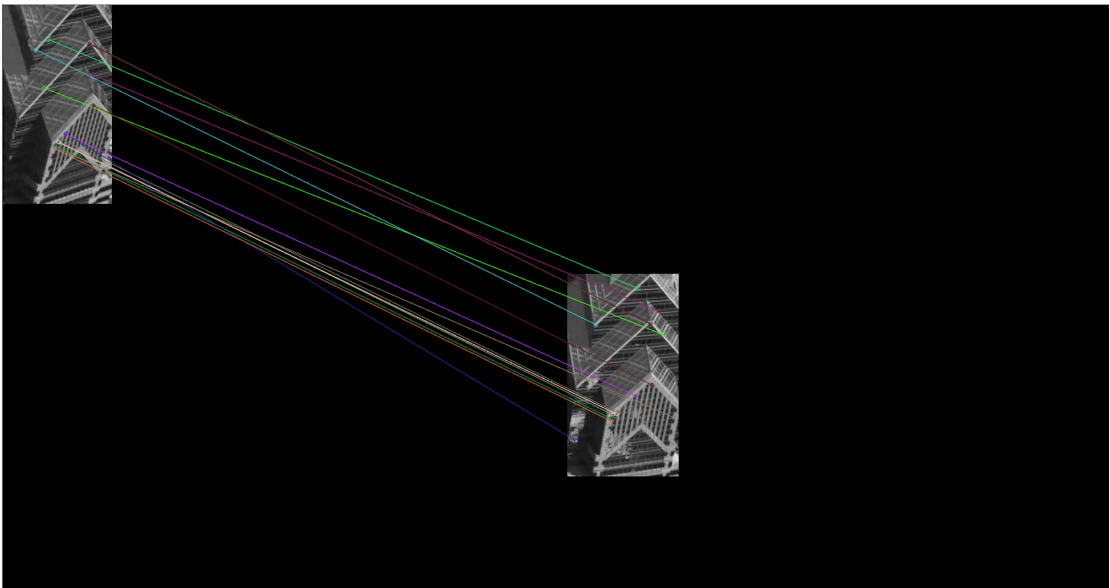

Figure 3.9 Key Point Matching with Masking

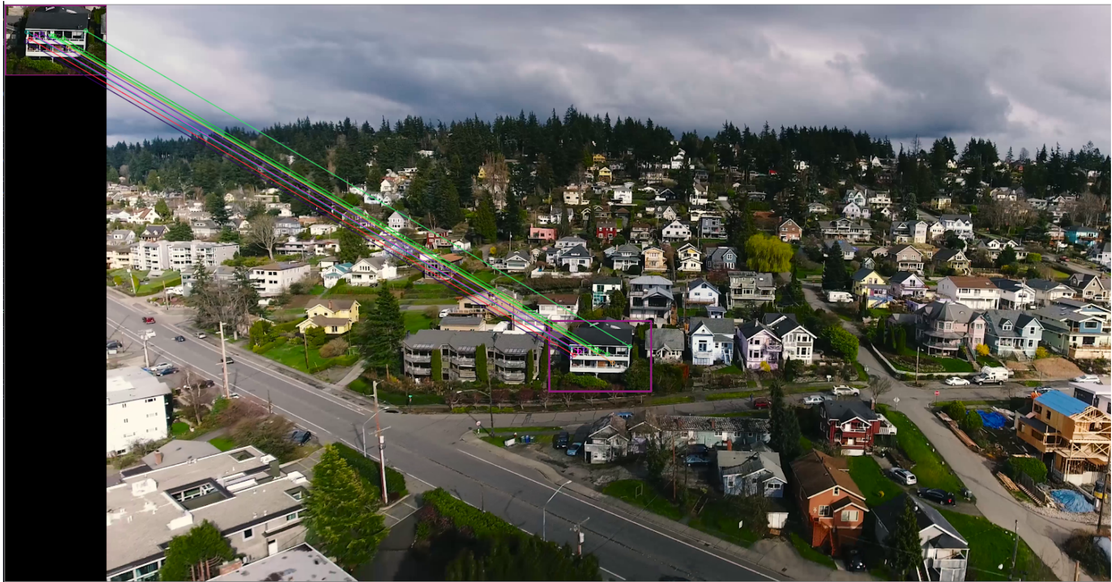

Figure 3.10 Masking with Original Frame Overlay

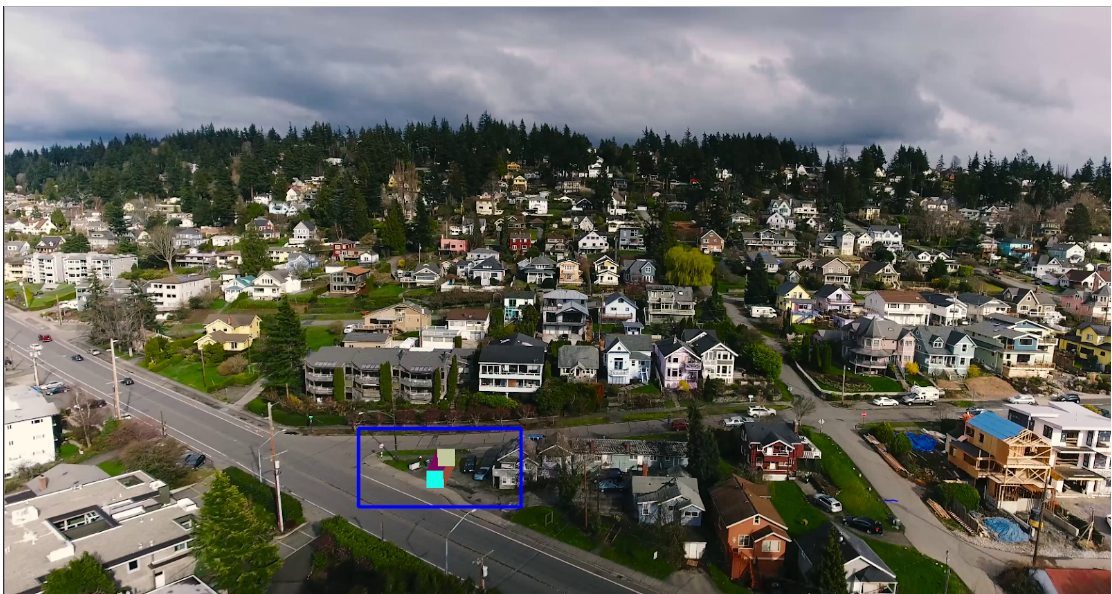

Figure 3.11 AR Projection

Figure 3.12 AR Over Time Position One

Figure 3.13 AR Over Time Position Two

Chapter 4

Discussion

4.1 The Future of Drone Based AR

This work successfully establishes the feasibility of drone based augmented reality, and it provides an introductory framework that can be built on for further applications. The results above demonstrate a working proof of concept for augmented reality from a drone video. The challenges facing Drone Based AR mentioned in the beginning can all be addressed by combining existing sensor processing and computer vision techniques.

4.2 Future Improvements

There are still many enhancements that could be implemented in a future project to enhance the functionality offered from this research. In order to speed up the programming process, python was used to implement this functionality, relying heavily on the OpenCV library. OpenCV is also an available package for the C++ programming language. C++ is more intensive upfront to develop, but it offers many performance increases over python. A future project could implement all the functionality from this project in C++ and achieve a noticeably higher frame-rate. Another possible future improvement would be new methods to instantiate the object in the frame. As discussed in the Results section, this research employs a user selected subset of the frame to serve as the model that the object is initialized on. Alternatively, this could be done using GPS or object recognition.

For example, a sensor could send its location to the server computer that hosted the drone feed, and the position it would be in the frame could be calculated based on the drone's GPS coordinates relative to the sensor. Alternatively, object recognition could be used such that the drone could display useful information in augmented reality about the object based on what it is recognized as. Another beneficial feature that could be implemented in further work is object scaling. This research anchor's the object in the frame and infers its proper orientation, but there is no run-time scaling of the object relative to the other objects in the frame. This research could also be extended to allow for more than one AR object to be created in the frame. Finally, in the future this program could be applied to a real time stream from a drone camera, and be able to provide relevant information to whoever is watching the feed.

Chapter 5

Conclusions

Augmented reality has widespread uses today, but there are very few cases where the technology is paired with drones. Drones are often deployed in complex environments, and they have extensive uses in commercial and recreational ways. Many drone use cases could be enhanced by the addition of Augmented reality. The main challenge with this is that traditional methods for AR are not sufficient when working with the dynamic videos coming from a mobile drone. When applied, these traditional methods would create an unacceptable user experience. This research extends these traditional techniques to make augmented reality a possibility for drone applications. Through the combination of existing methods, object tracking, and filtering algorithms, this project successfully adds an augmented reality layer to drone videos.

References

(1) Otto, A.; Agatz, N.; Campbell, J.; Golden, B.; Pesch, E. *Networks* **2018**, *72*, 411–458.

(2) Veas, E.; Grasset, R.; Kruijff, E.; Schmalstieg, D. *IEEE transactions on visualization and computer graphics* **2012**, *18*, 565–572.

(3) Ham, Y.; Han, K. K.; Lin, J. J.; Golparvar-Fard, M. *Visualization in Engineering* **2016**, *4*, 1.

(4) Marchand, E.; Uchiyama, H.; Spindler, F. *IEEE Transactions on Visualization and Computer Graphics* **2016**, *22*, 2633–2651.

(5) juangallostra Augmented Reality with Python and OpenCV `https://bitesofcode.wordpress.com/2017/09/12/augmented-reality-with-python-and-opencv-part-1/`, (accessed: 01.01.2020).

(6) Videezy.com 4K Aerial Drone Shot Of Liberty Place Buildings In Philadelphia `https://www.videezy.com/aerial-drone/11547-4k-aerial-drone-shot-of-liberty-place-buildings-in-philadelphia`, (accessed: 01.01.2020).

(7) Videezy.com Oregon Bridge River 4K Aerial Drone Shot `https://www.videezy.com/aerial-drone/10605-oregon-bridge-river-4k-aerial-drone-shot`, (accessed: 01.01.2020).

(8) Videezy.com Drone Footage Over Large Neighborhood `https://www.videezy.com/aerial-drone/7788-drone-footage-over-large-neighborhood`, (accessed: 01.01.2020).

(9) Pronios, F. AIPset1 `https://github.com/fpronios/AI_Pset1/blob/master/kalman.py`, (accessed: 01.01.2020).

(10) Open CV Documentation `https://docs.opencv.org/master/`, (accessed: 01.01.2020).

Chapter 6

Appendix

6.1 Full Python Source

```
from __future__ import print_function

import pandas as pd
import sys
import tkinter
import cv2 as cv
import PIL.Image, PIL.ImageTk
import time
import glob
import numpy as np
import matplotlib as plt
import math
import sys
from random import randint
# import Kfilter as k
import time
import random

#================================================================================
def kalman_xy(x, P, measurement, R,
              motion = np.matrix('0. 0. 0. 0.').T,
              Q = np.matrix(np.eye(4))):
    """
    Parameters:
    x: initial state 4-tuple of location and velocity: (x0, x1, x0_dot, x1_dot)
    P: initial uncertainty convariance matrix
    measurement: observed position
    R: measurement noise
    motion: external motion added to state vector x
    Q: motion noise (same shape as P)
    """
    return kalman(x, P, measurement, R, motion, Q,
                  F = np.matrix('''
```

```
                      1. 0. 1. 0.;
                      0. 1. 0. 1.;
                      0. 0. 1. 0.;
                      0. 0. 0. 1.
                      '''),
                  H = np.matrix('''
                      1. 0. 0. 0.;
                      0. 1. 0. 0.'''))

def kalman(x, P, measurement, R, motion, Q, F, H):
    '''
    Parameters:
    x: initial state
    P: initial uncertainty convariance matrix
    measurement: observed position (same shape as H*x)
    R: measurement noise (same shape as H)
    motion: external motion added to state vector x
    Q: motion noise (same shape as P)
    F: next state function: x_prime = F*x
    H: measurement function: position = H*x

    Return: the updated and predicted new values for (x, P)

    See also http://en.wikipedia.org/wiki/Kalman_filter

    This version of kalman can be applied to many different situations by
    appropriately defining F and H
    '''
    # UPDATE x, P based on measurement m
    # distance between measured and current position-belief
    y = np.matrix(measurement).T - H * x
    S = H * P * H.T + R  # residual convariance
    K = P * H.T * S.I    # Kalman gain
    x = x + K*y
    I = np.matrix(np.eye(F.shape[0])) # identity matrix
    P = (I - K*H)*P

    # PREDICT x, P based on motion
    x = F*x + motion
    P = F*P*F.T + Q
    return x, P

def demo_kalman_xy():
    x = np.matrix('0. 0. 0. 0.').T
    P = np.matrix(np.eye(4))*10
    R = 0.01**2
    N = 20
    observed_x = range(0, N)
```

```
    observed_y = [0]
    for i in range(1, N):
        observed_y.append(observed_y[i-1] + random.randint(1,4))


    plt.plot(observed_x, observed_y, 'ro')
    result = []

    for meas in zip(observed_x, observed_y):
        x, P = kalman_xy(x, P, meas, R)
        result.append((x[:2]).tolist())
    kalman_x, kalman_y = zip(*result)
    plt.plot(kalman_x, kalman_y, 'g-')
    plt.show()


class OBJ:
    def __init__(self, filename, swapyz=False):
        """Loads a Wavefront OBJ file. """
        self.vertices = []
        self.normals = []
        self.texcoords = []
        self.faces = []
        material = None
        for line in open(filename, "r"):
            if line.startswith('#'): continue
            values = line.split()
            if not values: continue
            if values[0] == 'v':
                v = [float(val) for val in values[1:4]]
                if swapyz:
                    v = v[0], v[2], v[1]
                self.vertices.append(v)
            elif values[0] == 'vn':
                v = [float(val) for val in values[1:4]]
                if swapyz:
                    v = v[0], v[2], v[1]
                self.normals.append(v)
            elif values[0] == 'vt':
                self.texcoords.append(map(float, values[1:3]))
            #elif values[0] in ('usemtl', 'usemat'):
                #material = values[1]
            #elif values[0] == 'mtllib':
                #self.mtl = MTL(values[1])
            elif values[0] == 'f':
                face = []
                texcoords = []
                norms = []
```

```
                for v in values[1:]:
                    w = v.split('/')
                    face.append(int(w[0]))
                    if len(w) >= 2 and len(w[1]) > 0:
                        texcoords.append(int(w[1]))
                    else:
                        texcoords.append(0)
                    if len(w) >= 3 and len(w[2]) > 0:
                        norms.append(int(w[2]))
                    else:
                        norms.append(0)
                #self.faces.append((face, norms, texcoords, material))
                self.faces.append((face, norms, texcoords))

MIN_MATCHES = 15
k_homography = np.zeros((3,3))
N = 6
p_homography = np.empty((10,3,3))
homography = None

camera_parameters = np.array([[1000, 0, 0], [0, 1000, 2000], [0, 0, 1]])

obj = OBJ('box/box.obj', swapyz=False)

screen_factor = 2.0

x = np.matrix('0. 0. 0. 0.').T
P = np.matrix(np.eye(4))*10
R = (0.01)**2

k_filters = [[{},{},{}],
             [{},{},{}],
             [{},{},{}]]
for i in range(3):
    for j in range(3):
        k_filters[i][j] = {'x':x, 'P':P,'R':R}


# Set video to load
# videoPath = "bridge.mp4"
videoPath = "city.mp4"
# videoPath = "houses.mp4"

skip_frames = 0

def projection_matrix(camera_parameters, homography):
    """
    From the camera calibration matrix and the estimated homography
```

```
    compute the 3D projection matrix
    """
    # Compute rotation along the x and y axis as well as the translation
    # homography = homography * (-1)
    rot_and_transl = np.dot(np.linalg.inv(camera_parameters), homography)
    col_1 = rot_and_transl[:, 0]
    col_2 = rot_and_transl[:, 1]
    col_3 = rot_and_transl[:, 2]
    # normalise vectors
    l = math.sqrt(np.linalg.norm(col_1, 2) * np.linalg.norm(col_2, 2))
    rot_1 = col_1 / l
    rot_2 = col_2 / l
    translation = col_3 / l
    # compute the orthonormal basis
    c = rot_1 + rot_2
    p = np.cross(rot_1, rot_2)
    d = np.cross(c, p)
    rot_1 = np.dot(c / np.linalg.norm(c, 2) + d / np.linalg.norm(d, 2), 1 / mat
    rot_2 = np.dot(c / np.linalg.norm(c, 2) - d / np.linalg.norm(d, 2), 1 / mat
    rot_3 = np.cross(rot_1, rot_2)
    # finally, compute the 3D projection matrix from the model to the current f
    projection = np.stack((rot_1, rot_2, rot_3, translation)).T
    return np.dot(camera_parameters, projection)

def render(img, obj, projection, model, color=False, colors=[]):
    vertices = obj.vertices
    scale_matrix = np.eye(3) * 25
    h, w, im_slice = model.shape

    for i, face in enumerate(obj.faces):
        face_vertices = face[0]
        points = np.array([vertices[vertex - 1] for vertex in face_vertices])
        points = np.dot(points, scale_matrix)
        # render model in the middle of the reference surface. To do so,
        # model points must be displaced
        points = np.array([[p[0] + w / 2, p[1] + h / 2, p[2]] for p in points])
        dst = cv.perspectiveTransform(points.reshape(-1, 1, 3), projection)
        imgpts = np.int32(dst)
        if color is False:
            cv.fillConvexPoly(img, imgpts, (211, 27, 30))
        else:
            color = colors[i]
            cv.fillConvexPoly(img, imgpts, color)
    return img

trackerTypes = ['BOOSTING', 'MIL', 'KCF','TLD', 'MEDIANFLOW', 'GOTURN', 'MOSSE'

def createTrackerByName(trackerType):
```

```
    # Create a tracker based on tracker name
    if trackerType == trackerTypes[0]:
        tracker = cv.TrackerBoosting_create()
    elif trackerType == trackerTypes[1]:
        tracker = cv.TrackerMIL_create()
    elif trackerType == trackerTypes[2]:
        tracker = cv.TrackerKCF_create()
    elif trackerType == trackerTypes[3]:
        tracker = cv.TrackerTLD_create()
    elif trackerType == trackerTypes[4]:
        tracker = cv.TrackerMedianFlow_create()
    elif trackerType == trackerTypes[5]:
        tracker = cv.TrackerGOTURN_create()
    elif trackerType == trackerTypes[6]:
        tracker = cv.TrackerMOSSE_create()
    elif trackerType == trackerTypes[7]:
        tracker = cv.TrackerCSRT_create()
    else:
        tracker = None
    return tracker

def img_resize(img, factor):
    height, width, layers =  img.shape
    new_h= int(height / factor)
    new_w= int(width / factor)
    return cv.resize(img, (new_w, new_h))

# Create a video capture object to read videos
cap = cv.VideoCapture(videoPath)

bboxes = []
colors = []
obj_colors = []

for i in range(6):
    obj_colors.append((randint(0, 255), randint(0, 255), randint(0, 255)))

frame_count = 0

success, frame = cap.read()
frame = img_resize(frame, screen_factor)
frame_count += 1

if not success:
    print('Failed to read video')
    sys.exit(1)

bbox = cv.selectROI('MultiTracker', frame)
```

```
bboxes.append(bbox)
colors.append((randint(0, 255), randint(0, 255), randint(0, 255)))


print('Selected bounding boxes {}'.format(bboxes))


trackerType = "CSRT"

multiTracker = cv.MultiTracker_create()

for bbox in bboxes:
    multiTracker.add(createTrackerByName(trackerType), frame, bbox)

found_count = 0
first_frame = True
orb = cv.ORB_create()
# Process video and track objects
start = time.monotonic()
while cap.isOpened():
    success, frame = cap.read()
    frame_count += 1
    if not success:
        break

    frame = img_resize(frame, screen_factor)
    gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
    success, boxes = multiTracker.update(frame)


    for i, newbox in enumerate(boxes):
        p1 = (int(newbox[0]), int(newbox[1]))
        p2 = (int(newbox[0] + newbox[2]), int(newbox[1] + newbox[3]))
        m1 = int(newbox[1])
        m2 = int(newbox[1] + newbox[3])
        m3 = int(newbox[0])
        m4 = int(newbox[0] + newbox[2])
        cv.rectangle(frame, p1, p2, colors[i], 2, 1)
        if first_frame:
            model_base = gray[m1:m2,m3:m4]
            c_model_base = frame[m1:m2,m3:m4]
        model = gray[m1:m2,m3:m4]
        c_model = frame[m1:m2,m3:m4]

        f_mask = np.zeros(gray.shape,np.uint8)
        f_mask[m1:m2,m3:m4] = gray[m1:m2,m3:m4]
        c_mask = np.zeros(frame.shape,np.uint8)
        c_mask[m1:m2,m3:m4] = frame[m1:m2,m3:m4]
```

```
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

kp_model, des_model = orb.detectAndCompute(model_base, None)
kp_frame, des_frame = orb.detectAndCompute(f_mask, None)

matches = bf.match(des_model, des_frame)
matches = sorted(matches, key=lambda x: x.distance)

h, w = model.shape

if len(matches) > MIN_MATCHES:
    # differenciate between source points and destination points
    src_pts = np.float32([kp_model[m.queryIdx].pt for m in matches]).reshape
    dst_pts = np.float32([kp_frame[m.trainIdx].pt for m in matches]).reshape
    # compute Homography
    homography, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 1.0)

    if first_frame:
        k_homography = homography
        first_frame = False

    # # KFILTER IMPLEMENTATION OPTIONAL
    # for i in range(0,3):
    #     for j in range(0,3):
    #         x_u, P_u = kalman_xy(
    #             k_filters[i][j]['x'],
    #             k_filters[i][j]['P'],
    #             (frame_count, homography[i][j]),
    #             k_filters[i][j]['R'])
    #         k_filters[i][j]['x'] = x_u
    #         k_filters[i][j]['P'] = P_u
    #         k_homography[i][j] = x_u[1]


    if frame_count > N:
        p_homography = np.concatenate((p_homography, [homography]),axis=0)
        p_homography = np.delete(p_homography,0,0)
        smooth_homography = p_homography.mean(axis=0)
    else:
        p_homography = np.concatenate((p_homography,[homography]),axis=0)
        smooth_homography = homography



    h, w = model.shape
```

```
        pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
        dst = cv.perspectiveTransform(pts, smooth_homography)

        if homography is not None:
            projection = projection_matrix(camera_parameters, smooth_homography)
            frame = render(frame, obj, projection, c_model, True, obj_colors)
        # # Show matching explicitly
        # frame = cv.drawMatches(model_base, kp_model, f_mask, kp_frame, matches

    cv.imshow('MultiTracker', frame)
    if cv.waitKey(1) & 0xFF == 27:  # Esc pressed
        break


end = time.monotonic()
total = end - start
framerate = frame_count/total
print('Frame rate: {} f/s'.format(framerate))
```