Computer Vision Project: Pool Ball Tracking

Submitted to: Dr. Numan Khurshid

Group Members

Name	CMS ID	Email
Syed Zain Abbas	0364297	sabbas.msee21seecs@seecs.edu.pk
Taimoor Hasan Khan	0360625	tkhan.msee21seecs@seecs.edu.pk
Yaseen Athar	0330327	yathar.msee20seecs@seecs.edu.pk

Table 1: Group YTZ information.

1. Project Description

The goal of this project was to be able to detect and show the movement of the pool balls moving on a pool table and deflecting with the table walls. For this purpose, two Android cameras were setup as IP webcams using the Android application *IP Webcam* by Pavel Khlebovich, and placed at different angles.

Figure 1: Illustration of Experimental Setup

The live camera feed was taken using the IP stream URL and fed to a Python script from which the video stream was processed, the two video frames were merged via stitching, pool ball detection was applied and final results displayed on single stream. An offline mode was also added.

The motivation of this project was to be able to learn and practically implement important concepts of computer vision such as homography computations, perspective transforms, image registration and stitching, learning-based object detection/tracking, motion detection and basic image processing. These concepts apply to a wide range of applications, including sports broadcasts, multi-camera surveillance, traffic flow management etc.

We were also focused on learning the software that is used to implement all these calculations and transformations. We have utilized Python libraries (numpy, multiprocessing etc.) extensively, along with source compiled OpenCV (dnn, features2D, imageproc etc.).

2. Description of Relevant Sources/Literature

Feature	Link to Source	Framework
Multi Camera Setup	https://docs.python.org/3/library/multiprocessing.html	Python Multiprocessing Module
Object Detection	(CUDA Acceleration) https://medium.com/analytics-vidhya/build-opencv-from-source-with-cuda-for-gpu-access-on-windows-5cd0ce2b9b37 https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html#install-zlib-windows (Installation after OpenCV with CUDA) https://www.youtube.com/watch?v=WK_2bpWj35A (Training) https://github.com/AlexeyAB/darknet (Labelling) https://tzutalin.github.io/labelImg/	Darknet YOLOv4 (Tiny) Nvidia CUDA
Image Stitching	(Framework Documentation) https://docs.opencv.org/3.4/da/d9b/group__features2d.html (Blogpost on Framework) https://towardsdatascience.com/image-panorama-stitching-with-opencv-2402bde6b46c (Blogpost on Video Stitching) https://www.pyimagesearch.com/2016/01/25/real-time-panorama-and-image-stitching-with-opencv/ (Blogpost on High Level API Stitching) https://www.pyimagesearch.com/2018/12/17/image-stitching-with-opencv-and-python/	OpenCV Features2D Module for Feature Matching
Top View Generation	(Top View) https://www.youtube.com/watch?v=Tm_7fGolVGE (Mouse Callbacks) https://www.youtube.com/watch?v=a7_dBO3EAng	OpenCV Mouse Callback and Perspective Functions
Heatmap Generation	(Background Subtraction) https://www.youtube.com/watch?v=eZ2kDurOodI https://pysource.com/2018/05/17/background-subtraction-opencv-3-4-with-python-3-tutorial-32/ (Heatmap after Background Subtraction) https://towardsdatascience.com/build-a-motion-heatmap-videousing-opencv-with-python-fd806e8a2340	OpenCV Basic Image Processing

Table 2: References and sources used in developing this project.

Note that OpenCV does not have the best documentation, so we often had to look at various other sources to be able to understand the module properly. However, the multiprocessing module was well-documented and proved sufficient for our purposes.

Further, note that CUDA acceleration and YOLO installation was probably the hardest part of the setup as the process is not very well documented at one place, and requires a lot of pathing manipulations. We ended up installing YOLO before OpenCV, and realized after having implemented YOLO that OpenCV needs to be built from source, with its *opencv-contrib* extension in order to use CUDA acceleration. We have included the links we most closely followed, but note that the 'medium' link misses a particular checkbox when building OpenCV with CMake GUI which we are unable to recall as of this writing.

In general, make sure to check every single option with CUDA and GPU in it if it applies to your particular case, and look up what they do before you check them. Installing Zlib is VERY IMPORTANT, and no tutorial mentions this. The installation guide we have mentioned follows the actual installation method proposed on AlexeyAB's Darknet repository for CMake installation of YOLO and also builds Darknet with CUDA. However, the guide does not build OpenCV with CUDA, so it is necessary you build OpenCV using the guides before it if you want to be able to use GPU within Python scripts and not just on the command-line.

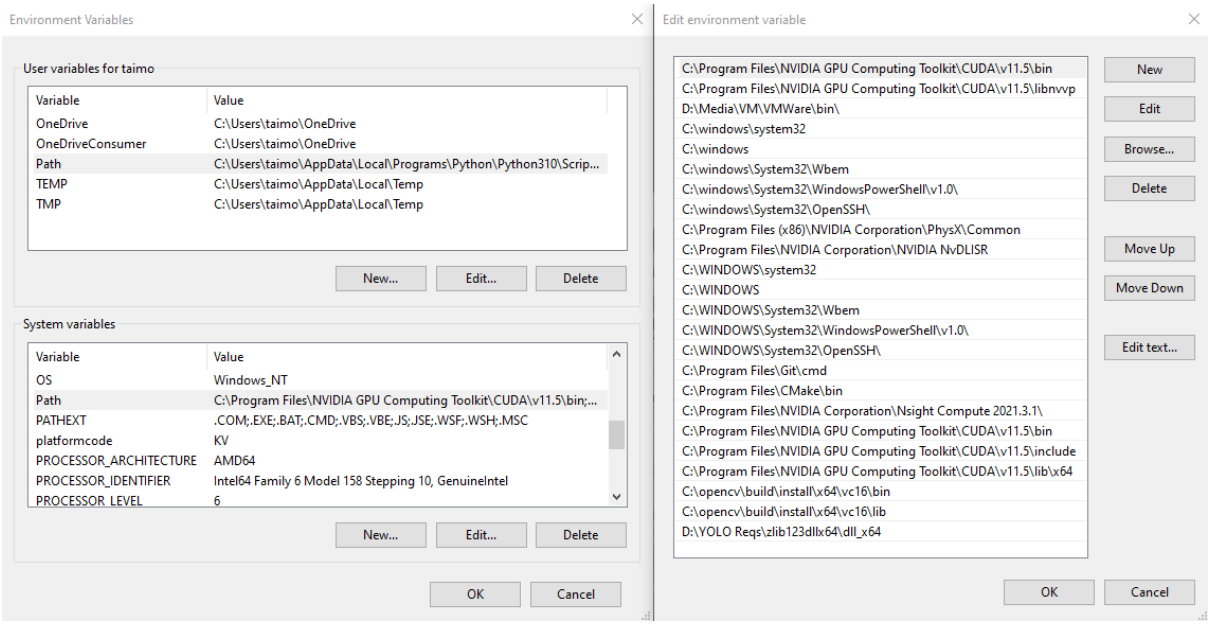


Figure 2: The path modifications needed for OpenCV and YOLO with CUDA.

For training our YOLO network, we have again utilized AlexeyAB's fork of Darknet, which provides very clear instructions on how to train the network using simple command line interfacing. However, note that the instructions provide paths for the now-defunct installation method of YOLO and need editing for CMake installation. General rule: follow AlexeyAB's method, but do it in the directory where your *darknet.exe* is (post-compile) and NOWHERE ELSE.

3. Planned Tasks with Learning Objectives

- Task 1: Multi Camera Setup and Recording

  The learning objective of this task was to become familiar with Python, any one of the many IDEs for Python development, OpenCV and finally to learn how to access multiple live camera feeds over a network (i.e., IP streams).

- Task 2: Object Detection

  The learning objective of this task was to setup the YOLO object detection network and train it for detecting images of a particular object (pool balls in our case) and then use it for inference in some application (pool ball tracking in our case).

- Task 3: Image Stitching (Video Stitching by extension)

  The learning objective of this task was to find the homography between images taken of the same scene from two different angles and apply image stitching for merged display. In our case, the concept extended to stitching two video feeds together, which presented a real-time operational challenge.

- Task 4: Orthographic Top-View Generation

  The learning objective of this task was somewhat similar to the stitching one, but this time with only four points on an image to generate a perspective transformation to the plane defined by these four points (bird's eye or top view).

- Task 5: Heatmap Generation

  The learning objective of this task was to implement a motion-based heatmap on the generated top-view in the previous task.

4. Methodology

An overview is given at the end of this section, if you do not wish to read all the details.

First of all, we acquired a small-scale pool table with a complete set of pool balls was acquired in order to have an easily accessible portable model for testing. We did not gather tripods for our cameras, opting to use stacked books and placing the cameras on top of them instead to record the feeds.

Task 1

Once the experimental setup was ready, we began work on the software end. Initially, we simply initialized three cameras within a script using Python and OpenCV. However, for more than one IP camera at any given time, the recordings were extremely laggy and all over the place (heavily buffered). To remedy this, we created a multi-threading setup to grab videos from three IP cameras using Python and OpenCV. This seemed to remove the lag completely.

However, as soon as we began any kind of image processing on any of the threads, the lag appeared once again and it was particularly strange because now, when one camera worked, the others froze. Then, the second camera would work, and the first and third would freeze. We concluded after reading up some source that this was because multi-threading only

guarantees concurrency, but it not truly parallel processing. It certainly explained the one-at-a time behavior we were observing here.

To fix this new problem, we tried implementing multiprocessing. At the time, were using JPNB and we learnt the hard way that JPNB does NOT support Python's multiprocessing module (using the module requires that the child function can call the main function, and this is not supported in JPNB). The result was that the JPNB would just loop as 'busy' infinitely.

At this point, we moved the code to Visual Studio Code, and began using multiprocessing here. The difference in implementation between multi-threading and multiprocessing was not much, so this was easy enough to implement. After multiprocessing, we noticed that not only did the cameras work in perfect time with each other for just display, they also worked excellently during image processing, which concluded this task.

Task 2

Possibly the most arduous task of all, implementing YOLO for object detection is where we had the most of our adventures.

- Pre-Requisites and Setup

We had an Nvidia RTX 2060 GPU (compute capable rating of 7.5) in one of our group member's laptops (Taimoor), and he volunteered to utilize it for this project. Of course, this meant learning how to use CUDA with the OpenCV library and for YOLO as well. We had no idea about any of this, and made a large number of mistakes before finally achieving success. It was particularly painful because if the build failed for any reason, or the method used was now considered defunct but not explicitly mentioned in that one old tutorial, we had to restart from scratch to ensure a clean setup.

Building OpenCV with CUDA support takes 55 minutes (excluding download and installation times of pre-requisites) on an i7-9750H 6-Core @2.60 GHz, we can only imagine how long it takes on anything else! Also, note that in order to install OpenCV from source, we had to first remove the cv2 library for python and NEVER install it again (we did it by accident while working with Tensorflow), otherwise the path variables become messed up and Python decides to use cv2 rather than the source-compiled OpenCV, making it necessary to re-build.

- Pre-Trained COCO Results

Once we had finally managed to set it up properly and validated that the GPU was indeed being detected as a CUDA enabled device, we ran some tests on YOLO. Initially, we forgot to set the preferable backend targets to CUDA (required by OpenCV in order to use GPU), and were surprised to see 2 FPS at the output with COCO weights. We realized this quickly, and after setting the preferable backend to GPU, received roughly **~10 FPS** on average, and noticed the lag on the networks was back again. These results were with **YOLOv4**. After discussion with the instructor, we decided that **YOLOv4 Tiny** was sufficient for our purposes and would provide a significant boost to FPS, a focus of our real time application and decided to use that instead. For the pre-trained weights, we saw a boost all the way to **~30 FPS** and the network lag was also removed. Further we had some issues with getting the labels to draw properly on the image (the bbox would draw, but not the text inside it). We fixed these by following:

https://github.com/pjreddie/darknet/issues/955

```
267 image **load_alphabet()
268 {
269     int i, j;
270     const int nsize = 8;
271     image** alphabets = (image**)xcalloc(nsize, sizeof(image*));
272     for(j = 0; j < nsize; ++j){
273         alphabets[j] = (image*)xcalloc(128, sizeof(image));
274         for(i = 32; i < 127; ++i){
275             char buff[256];
276             sprintf(buff, "D:/YoloV4_DarkNet/darknet/data/labels/%d_%d.png", i, j);
277             alphabets[j][i] = load_image_color(buff, 0, 0);
278         }
279     }
280     return alphabets;
281 }
```

Figure 3: Highlighting the necessary change (line 276) to get labels to draw properly in *darknet/src/images.c*

- OpenCV and IP Webcams

This network lag that we observe is not a problem with the IP webcam or the network, it is with HOW OpenCV reads them. A USB webcam never lags when the FPS drop below the native FPS of the camera, but the same is not true for an IP webcam on OpenCV (darknet still works fine with IP cams; we verified this on command-line). We are not entirely sure, but it has something to do with how OpenCV buffers and decodes IP streams. As OpenCV was crucial for us, we needed to maintain the achieved FPS over the native FPS of the camera in order to achieve a real-time feed.

A possible alternative to this might be to use *imutils video.VideoStream* object to capture feeds instead, it is supposed to be better as it utilizes threading for streams internally, which ought to be very fast and reduce the decode lag on the stream. However, we did not test it. Another solution might be to use a queue-based synchronizer, where each camera reads a frame, puts it in a queue assigned to it, and waits until all the other cameras have also put a frame in their respective queues before reading the next frame. The same applies to all cameras. Regardless, using YOLOv4 Tiny removed this lag, so we did not pay it any further mind.

Note: Another anomaly we discovered in the process was that reducing the frame rate of the IP camera itself below the achieved FPS does NOT seem to help with the network lag either. It improves it at the start, but the buffer lag between the 2 cameras slowly increases as time progresses (i.e., cameras are in perfect sync with the world initially, and slowly the time gap between them increases). We were unable to pinpoint the cause for this issue; perhaps a hardware difference between the cameras (Redmi Note9S and PocoX3).

- Training

We gathered exactly 210 images of pool balls initially. Of these, around 30 were from our own setup at various orientations, lighting conditions, and pool ball positions. The rest of the 180 were scraped from ImageNet using:

https://github.com/skaldek/ImageNet-Datasets-Downloader

which is a forked version of:

https://github.com/mf1024/ImageNet-Datasets-Downloader

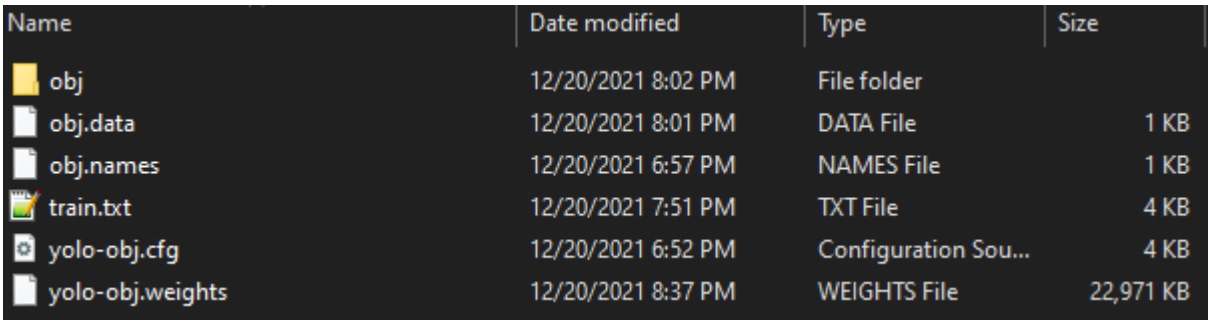
The reason we use the forked version is because the original was intended for Linux systems, and Linux has a different take on multiprocessing operations than Windows. Multiprocessing is used in the script, so obviously modifications are needed to get it to work on Windows. The forked repository contains these modifications and is verified to work. This issue with the script is popular in the "Issues" section of the original repository's Github.

We went through the samples at the end, and removed 3 blurry samples for a final total of 207. Note that almost all images had multiple instances of the object. LabelImg v1.8.0 for Windows was used to generate the text files containing the information on object locations within the images. These had to be manually generated and with care that the bounding box was as tight as possible around the object, and contained only one instance of the object. Therefore, this took a long time.

At the time, we were unaware of data augmentation sites such as Roboflow, that allow a variety of augmentation techniques specific to a particular network (including YOLO), which would have probably helped improve our results. Further note that we were also unaware of the concept of control samples at the time, otherwise we would have included images of objects that look like balls, but are not pool balls. AlexeyAB recommends having as many control samples as actual samples, so in our case we should have kept 207 images of similar looking objects, and augmented them the same way. Anyway, we proceeded with our sample set of 207 images with the following training parameters:

1. Classes = 1 (can be one or multiple)
2. Batch Size = 64 (total image sample set split into these batches)
3. Subdivisions = 16 (batches further split into these mini-batches)
4. Max Batches = 2000 (no. of iterations, basically)
5. Blob Size = 416, 416 (recommended for YoloV4 Tiny, can be any multiple of 32)

For the entire training process, we followed the instructions on AlexeyAB's darknet fork on Github, so we do not document the entire process here. We have included the script to generate a train.txt given the training images and their labels.txt software in the project's Github link (provided at the start of this document).



Name	Date modified	Type	Size
obj	12/20/2021 8:02 PM	File folder	
obj.data	12/20/2021 8:01 PM	DATA File	1 KB
obj.names	12/20/2021 6:57 PM	NAMES File	1 KB
train.txt	12/20/2021 7:51 PM	TXT File	4 KB
yolo-obj.cfg	12/20/2021 6:52 PM	Configuration Sou...	4 KB
yolo-obj.weights	12/20/2021 8:37 PM	WEIGHTS File	22,971 KB

Figure 4: All the files used for training the network and the results (weights) in one place. The folder 'obj' contains the 207 pool ball images with their label texts.

- Training Results

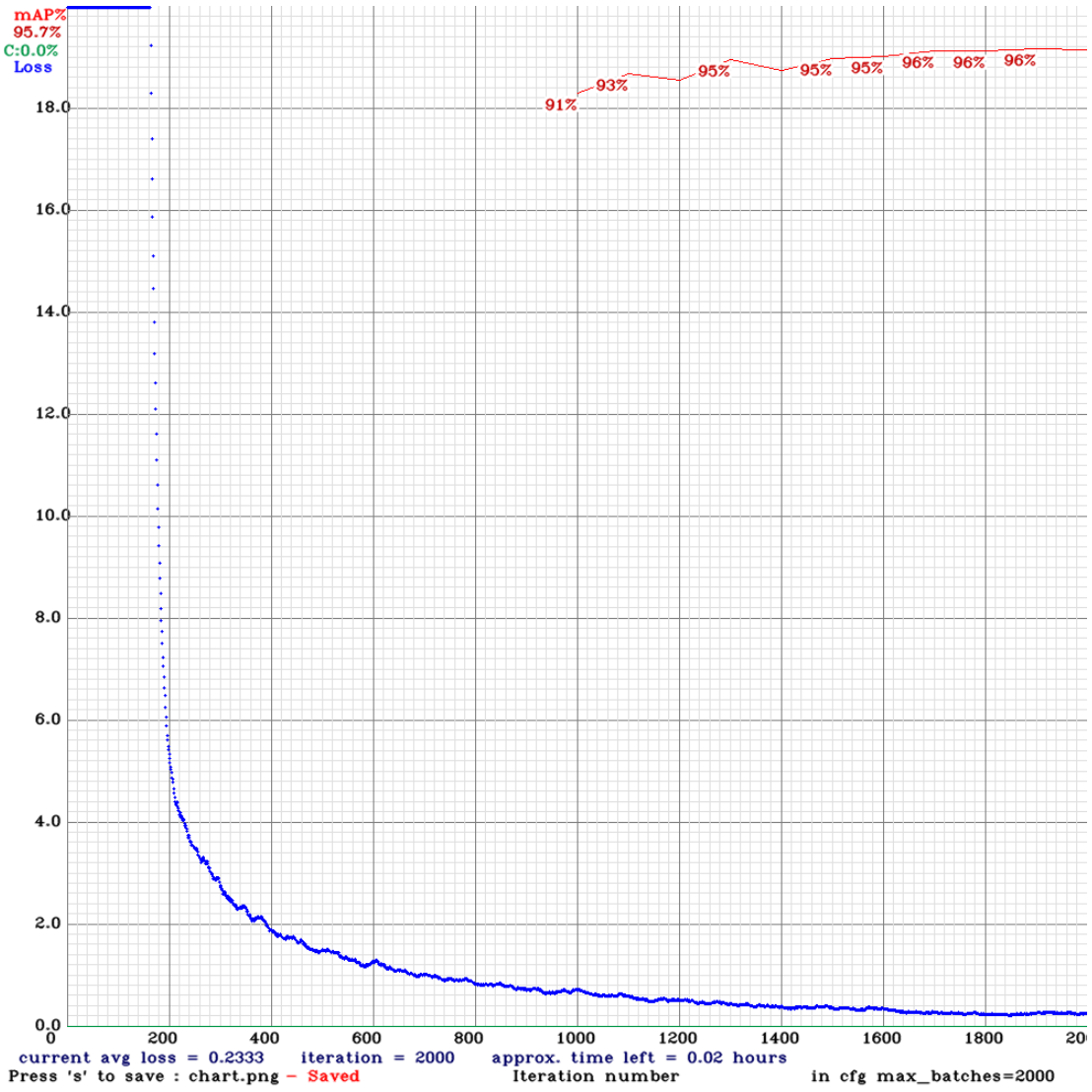


Figure 5: Training conclusion chart, as saved by Darknet. The red plot indicates Mean Average Precision (mAP, calculated every 100th iteration from the 1000th iteration onwards), and blue plot indicates the change in loss values. You can see it is very high at the start, followed by a large dip and smooth convergence to its final value.

The object detection was the verified on both static images and video streams using both CLI and an OpenCV script. These images were not used in training, albeit the pool table is the same.

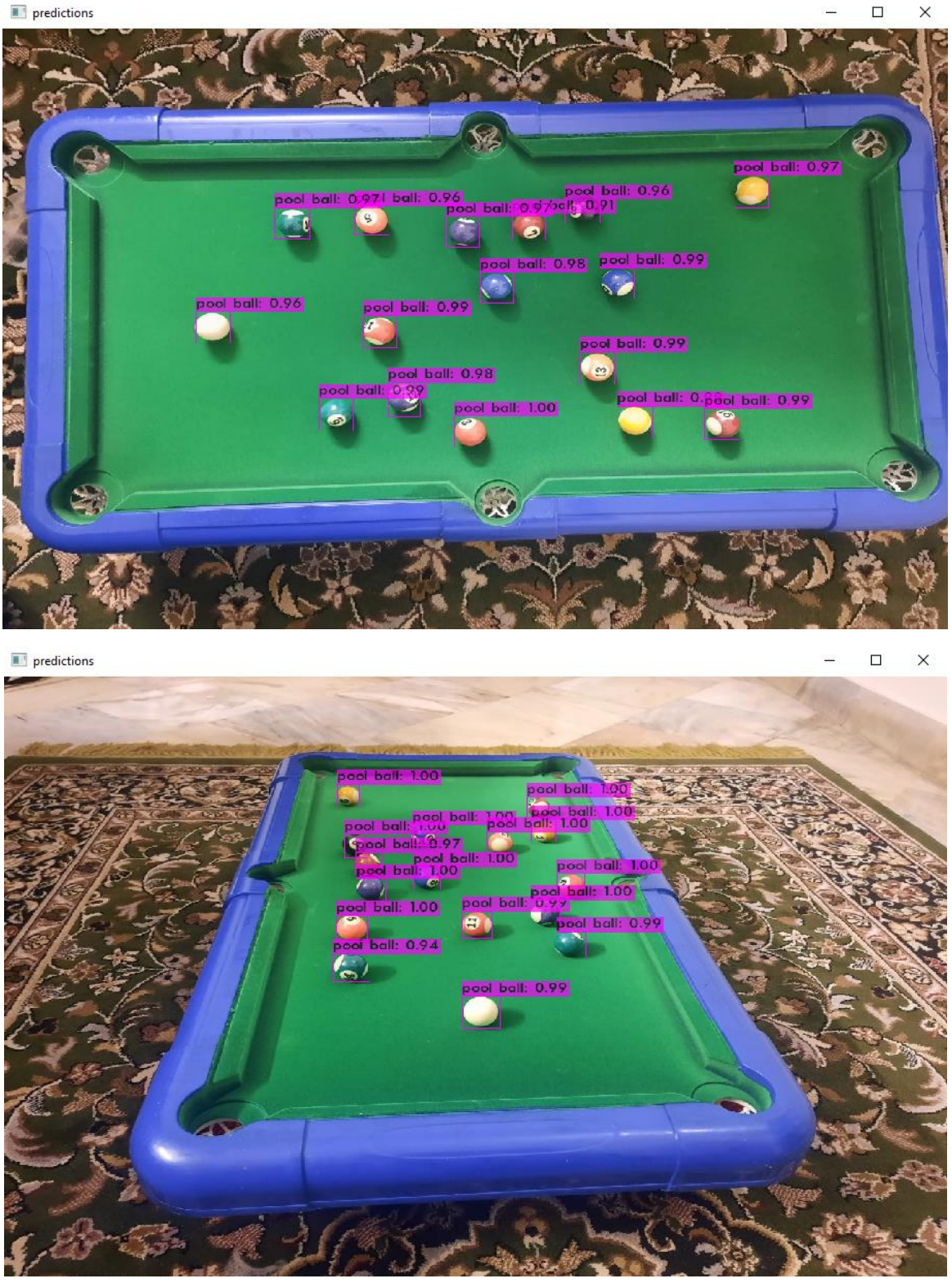


Figure 6: CLI results on training. OpenCV results not included because we have not used PyShine for drawing the textboxes which makes the text hard to read. CLI results have a letterbox by default, which is easier to read.

Task 3

Now, we needed to implement image stitching for videos. There were quite a few challenges, such as stitching 3 images (not just two) and doing so in real-time since stitching is a time-

consuming process. In our case, we also needed to figure out how to communicate between the stitcher process and the camera processes in order to synchronize frames from both the cameras in the multiprocessing framework. After discussion with the instructor, we reduced the number of cameras from 3 to 2, which greatly simplified things. However, we wrote the code so that it was expandable to 3 images regardless with some modifications.

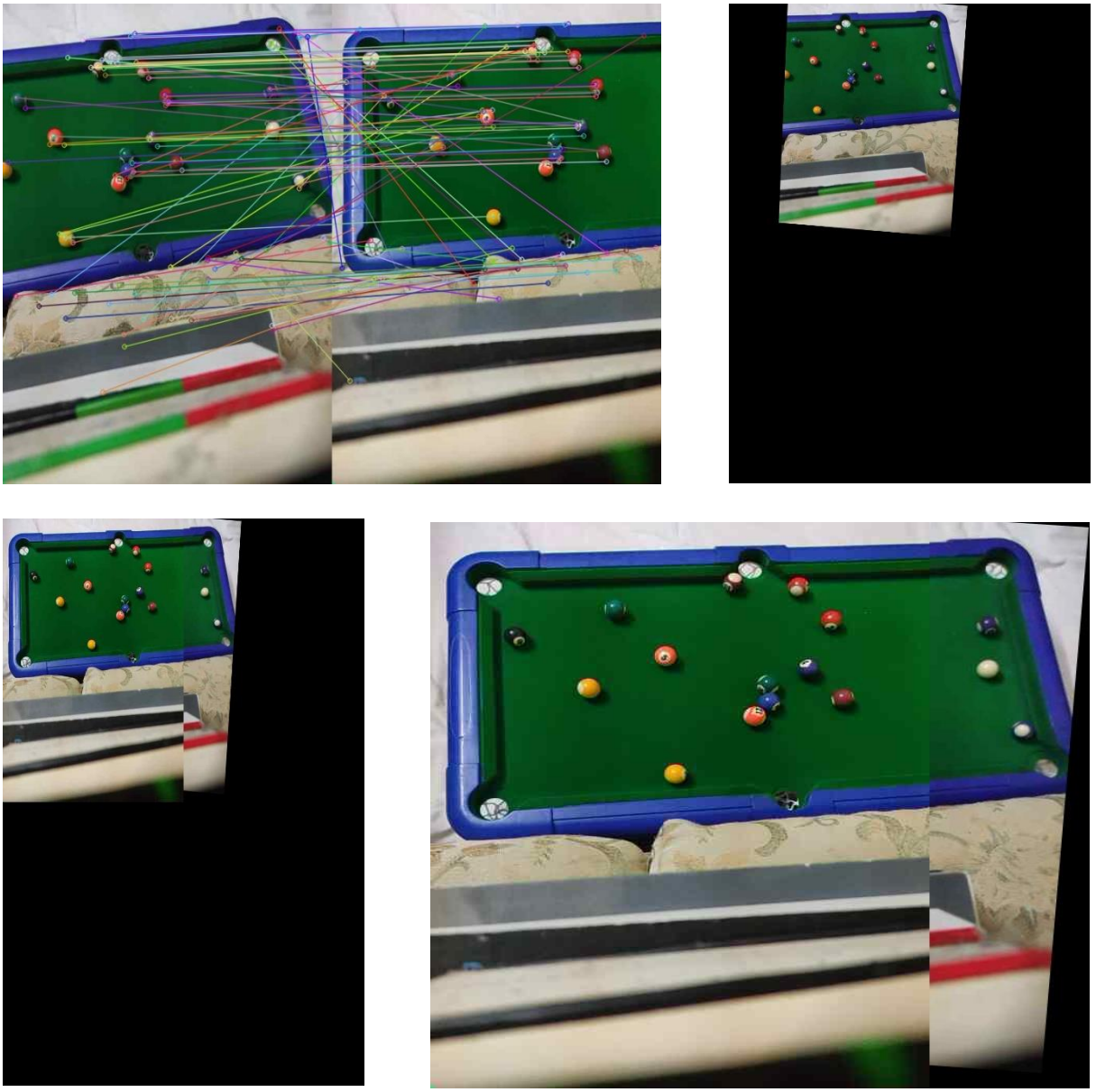


Figure 7: Stitcher outputs. Top to bottom, left to right: Matched features (first image is the right camera and second image is the left camera); warped right camera image; stitched image over largest canvas (simply overlayed left camera image over right); refined result.

We tried several different algorithms such as ‘ORB’, ‘SURF’ and ‘SIFT’. ‘ORB’ was seen to be the fastest, but it was generally unable to detect enough keypoints between the 2 camera images. By the end, we decided to use SIFT, which was giving us a stitched result in ~0.11 seconds on average (10 FPS) with K Nearest Neighbor (KNN) descriptor matching, which is less robust but fast.

In order to avoid the network lag that resulted from this, we made the assumption that our cameras would not move throughout the script's life. This assumption implied that we needed to compute the homography mapping the right camera image to the left only once. For the rest of the camera feeds, we can use the same homography to warp the two camera frames together! The warping in and of itself does not take much time, so this would boost our FPS.

Using the aforementioned logic with a simple if-elif section, we managed to get video stitching (for every 2 frames; one from each camera) results at ~**0.0030** seconds, which is a whopping **333 FPS**! Since we were performing stitching only once now, we decided to use the most robust methods we could: SIFT for keypoint generation with descriptors, descriptor matching with Brute Force (BF) and *crossCheck* flag set (details on Github, *img_stitching_detailed.py*). We refined this result a bit to reduce the amount of black regions in the final result after warping without losing information.

Further, we used a simple queue-event form of communication between the stitcher and camera processes, so that the cameras would sleep after reading a frame until the stitcher had successfully stitched them together, after which they would wake and read the next frame and so on. This ensured the frames remained synchronized and at any given time, the stitcher only received two frames.

Finally, homography estimation was done internally via RANSAC.

Tasks 4 & 5

Task 4 was quite similar to the task of stitching, except that now we had to use just 4 points marked on the planar object. In our case, we decided to use the table corners as they get us a good bird's eye view of the table. These marks are manually made by the user using OpenCV's high level GUI mouse callback function, which in this case returns the coordinates of the point the user left-clicks at. Of course, this is also done just once following the same stationary camera assumption as before. The top view homography is computed using these 4 marked points using OpenCV's *getPerspectiveTransform*() method, which is optimized for working with just 4 points unlike *findHomography*() method used for stitching, which is optimized for overconstrained systems (i.e., containing more than 4 point sets).

Task 5 simply expands upon the top view by performing background subtraction using a Mixture of Gaussians (MoG) model, which is a technique based on averaged frame differencing. The extracted foreground from this background is essentially a binary mask with whites on the foreground locations (movement detected w.r.t. average of last few frames). We can apply a colormap to this mask to end up with a motion heatmap over the complete history of the script's run-time or, more realistically, upto a last few seconds which is easier to read and store. More details can be found in the official OpenCV documentation, as well as the included scripts on the project's Github repo.

The results (top view and motion heatmap) are not shown here, as they are available in the recorded videos presented in the Github repo. You can record them yourself by pressing 'r' any time during the process as well.

The following diagram nicely summarizes our script's workflow.

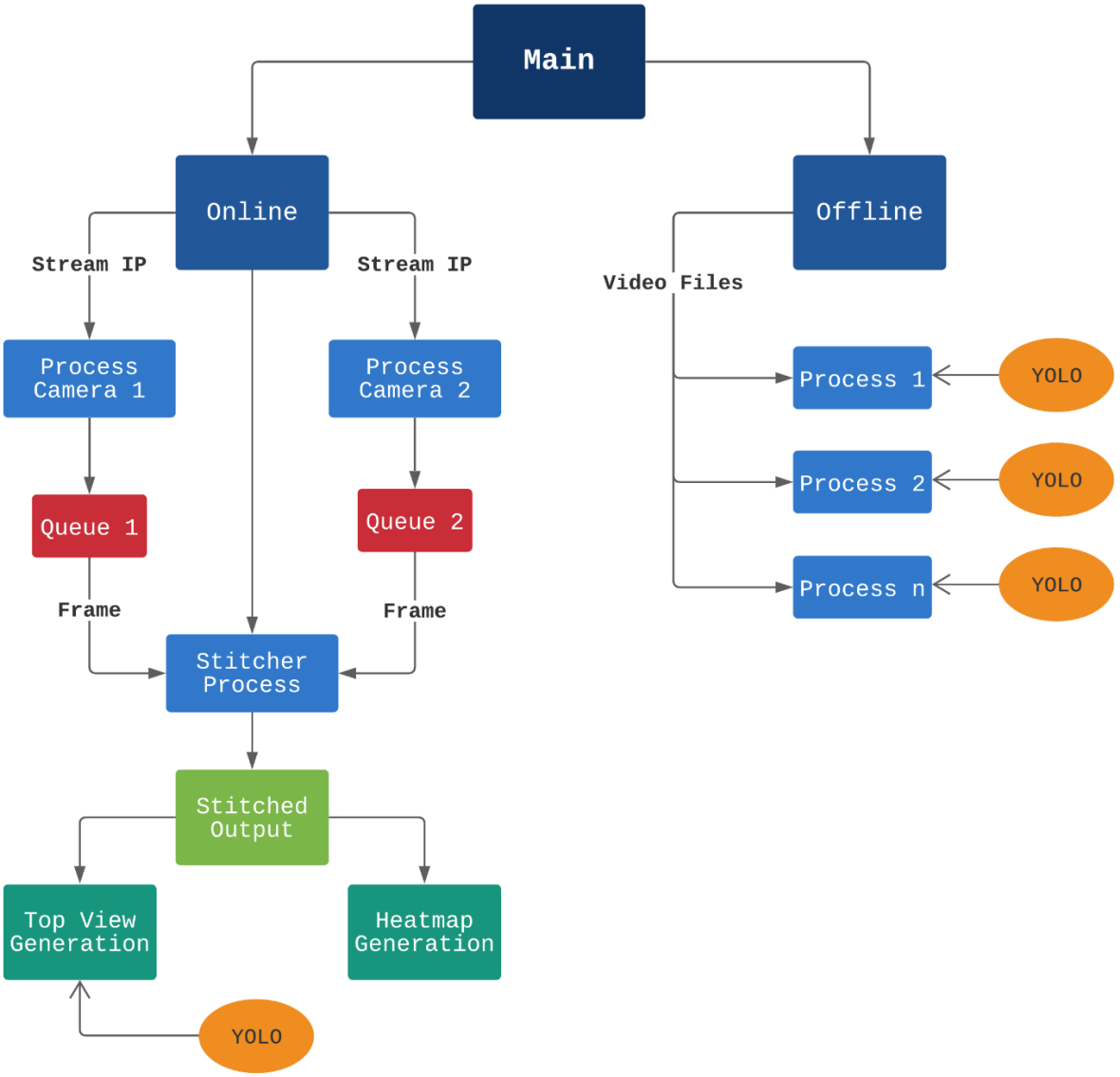


Figure 8: Multiprocessing framework and general application flow.

5. Realization of Theoretical Concepts

The theoretical concepts realized in this project are: Training Neural Networks for detecting required features, homography estimation for both fully and over-constrained systems, image stitching and other image processing techniques such as resizing, erosion-dilations, color space conversions, binary mask processing etc.

Learning Outcomes/Achievements

The learning outcomes and achievements are summarized below:

- Achievement 1: Successfully setup Python, OpenCV, and Darknet for YOLOv4 with Nvidia CUDA support.

  Skill: This taught us how to install Python and OpenCV and familiarized us with the various libraries and functions used in vision-based applications.

- Achievement 2: Created a multiprocessing framework and training YOLO with GPU acceleration for inference on pool balls.

  Skill: This taught us how to implement multi-threading and accelerating the application using GPU to have lag free implementation of application and reduce network training times.

- Achievement 3: Successfully used YOLO for object detection on Pool Balls in real-time.

  Skill: This taught us how to train Neural Networks (specifically YOLOv4 Tiny) in order to implement object detection on custom objects.

- Achievement 4: Successfully stitched images and videos in real-time.

  Skill: This taught us how to compute homography between two images using the overlapping image in the two and create a merged view.

- Achievement 5: Successfully generated a top view of the pool table and integrated various processing techniques in order to build the motion heatmap.

  Skill: Most importantly this project taught us how to bring all the core concepts together to create a working and useful application.

6. Completion Status

The project is complete with the five main deliverables and in working condition as detailed in above discussion. However, we do face occasional hiccups with the webcams being even slightly out of sync, which has a noticeable effect on the output. This effect is minimized when the heatmap is not being handled along with the stitch, so it may be better to record the top view only, and then construct a heatmap from the recorded top view locally for the most synchronized result. We have been unable to completely get rid of this problem as of now, though it has been minimized with the queueing structure we followed.

7. Future Extension

A future extension would be to implement a visual trip wire region which in this case would mark the area of the pockets which can be used to count how many pool balls have been pocketed.

Contributions of Members

1) Yaseen Athar: OpenCV Setup (Joint), exploring algorithms for homography computation and image stitching.
2) Taimoor Hasan Khan: OpenCV Setup (Joint), multiprocessing, GPU Acceleration, YOLO Object Detection.
3) Syed Zain Abbas: OpenCV Setup (Joint), implementing stitching on video streams.

8. Presentation Questions

We were unable to answer a few questions during our presentation with conviction. We will hopefully be able to resolve them here:

Q1: Why does YOLO need image sizes as multiples of 32 for its inferencing and training?

A (Presentation): Unable to answer with conviction, mentioned something about dividing the image into grids based on some visualization seen earlier when working on YOLO (Taimoor), as well as this was probably because YOLO has its layers defined with respect to the native resolution of 416 x 416.

A (Now): As per what was said during the presentation, YOLO indeed divides an incoming image into sub-grids (anchors) and then builds probability maps within those anchors. Different YOLO versions may do this at multiple scales (e.g. v3 and v4, but not v2 and v1). However, this gridding does not explain why it needs a multiple of 32x32 image size, for which the best explanation we have been able to find is that YOLO uses random distributions of images of the input at multiple scales (multiples of 32) during training, and therefore the input must be divisible by 32. It's just how it is built to handle input within its layers. Darknet automatically handles this resizing during training and OpenCV's DNN module is also capable of doing the same while preserving aspect ratio.

Q2: How does YOLO return bounding boxes (bboxes) on to the original image when it has resized the image?

A (Presentation): Would have to review code, knew at the time of implementation (Taimoor). Also mentioned about Darknet and OpenCV's DNN handling the resizing, so the user doesn't really need to worry about this.

A (Now): We had actually tested this at the time we were implementing YOLO (we had commented the logic back then), but essentially YOLO handles image coordinates as NORMALIZED image coordinates. This means, for an image of 640 x 480, the standard center point is considered at 320 x 240, but YOLO sees this as (0.5, 0.5). The bbox is also defined in these normalized terms, including its width and height; representing what percentage of the image is occupied by the bbox. Since the resizing (preserving aspect ratio) is done internally by the OpenCV and Darknet modules, the modules have access to the exact scaling factors to rescale these bboxes back to the original input size while preserving original aspect ratio, and this is why the bboxes in the final output are not messed up!

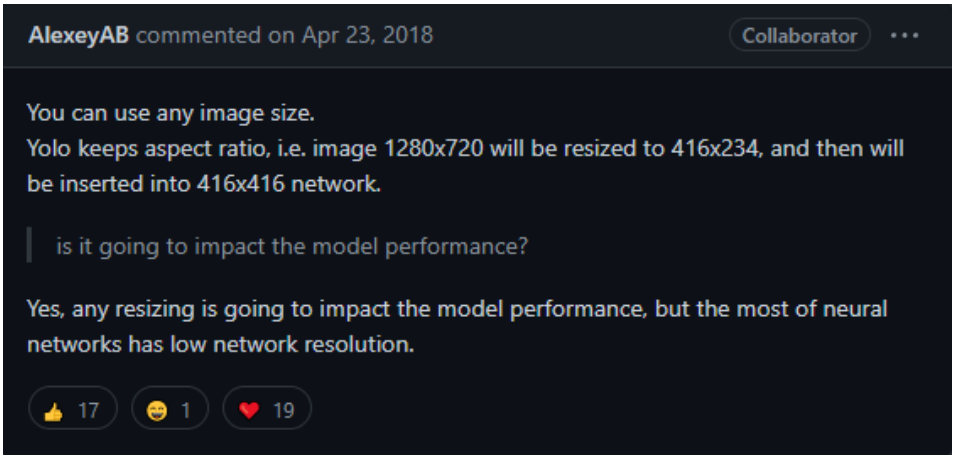


Figure 9: The man himself.