Table of Contents

Author: Gaurav Mohan

Developing a Basketball Minimap for Player Tracking using Broadcast Data and Applied Homography

1. Introduction

1.1 Concept

Player Tracking has become a big phenomenon in sports as it allows data scientists to make more in-depth analysis on a player's impact based on how the player moves around the court. For instance, teams can merge play-by-play data and player tracking simulations to add context to a player's impact when on the court. Now, if you take a player like Draymond Green you can visualize his ability to clog up cutting lanes, get out in transition, and set effective screens. These simulations can be much easier to visualize on a two-dimensional level compared to actual game film where broadcast angles can't capture spacing on the court as effectively.

1.2 Use Cases

This research project is focused on exploring applications of homography on broadcast footage of basketball games to create a two-dimensional simulated minimap. A homography is a type of projective transformation in that we take advantage of projections to relate two images. In essence, a homography is a transformation between two images of the same scene, but from a different perspective. While the NBA captures images from numerous different locations to pinpoint a player on the court, many high school games and some collegiate games are only captured from a single-angled broadcast view. Thus, I wanted to explore the effectiveness of applying homography on broadcast footage to track player movement on the court. If teams can use open source technology to create these simulations, then high school basketball programs can truly help young players work on their strengths and weaknesses at an early age, by leveraging these minimaps to measure more advanced analytics and add context to player impact.

2. Data Sources and Tools

2.1 Data Sources

The input video is taken from a Warriors vs Lakers game in the 2021-2022 NBA season. The snippet is about 40 seconds long and is sourced from YouTube.

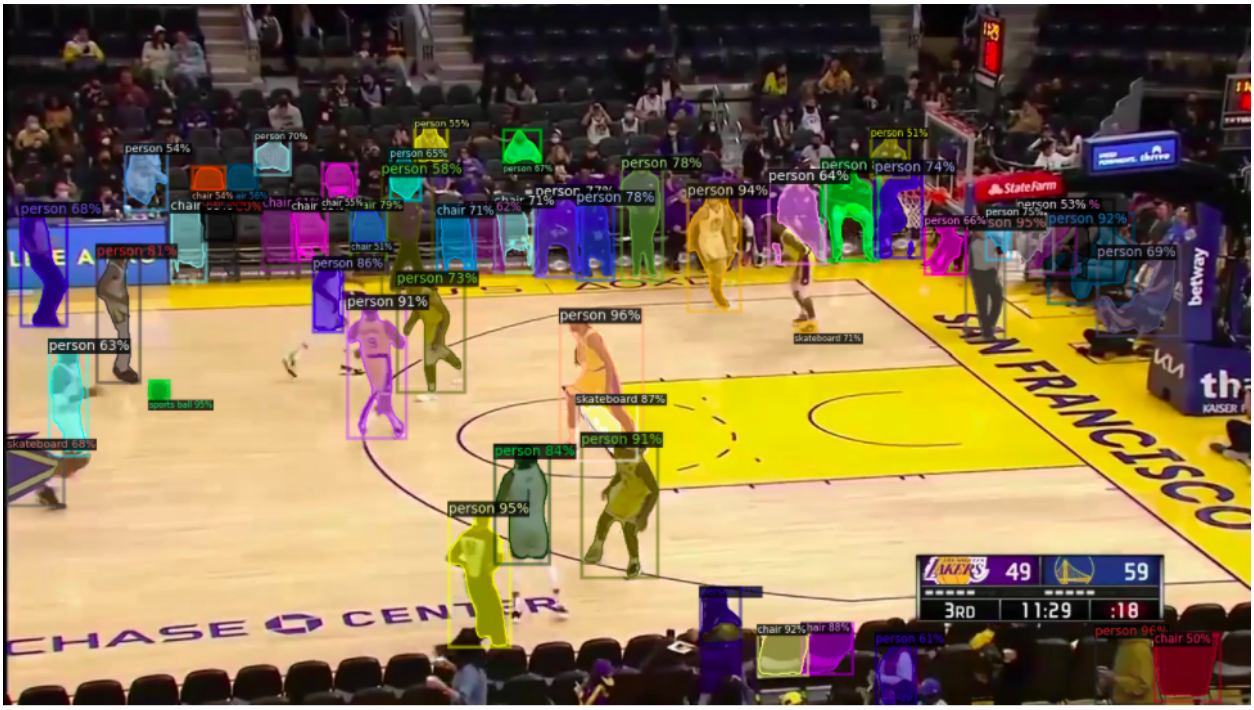
2.2 Tools

This project relied on a number of computer vision libraries within python. Pytorch is used to train the COCO dataset with detectron2. COCO dataset has pretrained objects that are used to recognize players and the basketball within each frame of the video, while detectron2 is the detectron algorithm that utilizes COCO. Shapely is used to pinpoint player objects that are within the boundaries of the court. Shapely is a Python package for manipulation and analysis of planar geometric objects. OpenCV is also used along with google colab to analyze frames and apply our homography function while also writing our output images post-application to our output video.

3. Pre-Processing

3.1 Detection Constraints

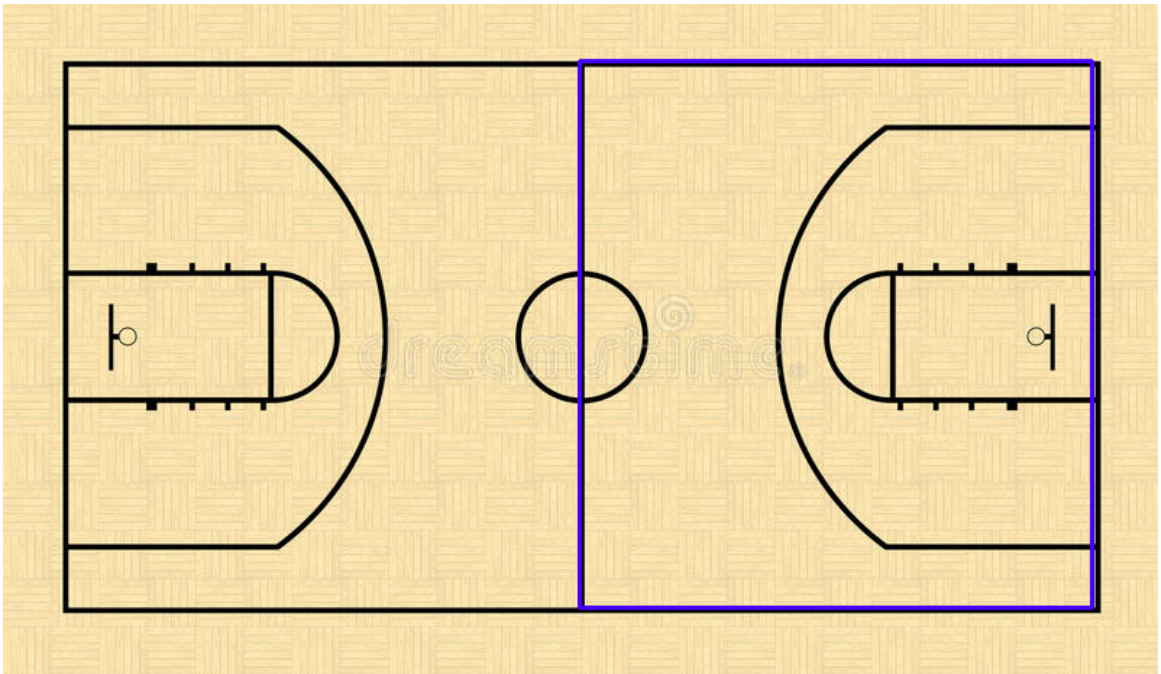
The process for developing the minimap starts with applying our detection algorithm on each frame from the input video to detect both the players on the court and the ball. When the detectron2 visualizer is applied on one of the initial frames from an NBA game the image is shown below.



Now that detectron2 can detect both the ball and the players, I need to reduce the noise by only capturing players that are within the court. The next step is to

draw visual boundaries on our image and map it onto a 2D basketball court that will be used for our minimap. The idea is to apply our object detection within the boundaries of the court so that we are only tracking legitimate players and not fans, coaches, or referees on the sideline.

From here I applied the homography function onto a set of points to convert a 3D vector into a 2D point within this specified zone on the court image.



3.2 Homography Transformation

The detectron2 algorithm utilizes a DefaultPredictor.predictor method to return a list of rectangle coordinates (pred_boxes) of each identified object. The object classes are stored in pred_classes, where player objects are marked as 0 while a ball is marked as 32. For each predicted object that is a player or a ball, the dimensions of the prediction box is stored in a dictionary along with the transformed data point that represents the position of the player. The transformed data point is created by converting the three-dimensional vectors representing players into a 2xN array representing player positions. This function is shown below.

```
def apply_homography(H,pts):
    """
    Apply a specified homography H to a set of 2D point coordinates

    Parameters
    ----------
    H : 3x3 array
        matrix describing the transformation

    pts : 2xN array
        2D coordinates of points to transform

    Returns
    -------
    numpy.array (dtype=float)
        2xN array containing the transformed points

    """

    #assert expected dimensions of input
    assert(H.shape==(3,3))
    assert(pts.shape[0]==2)
    assert(pts.shape[1]>=1)

    tpts = np.zeros(pts.shape)
    for i in range(pts.shape[1]):
        u = H[0][0]*pts[0][i] + H[0][1]*pts[1][i] + H[0][2]
        v = H[1][0]*pts[0][i] + H[1][1]*pts[1][i] + H[1][2]
        w = H[2][0]*pts[0][i] + H[2][1]*pts[1][i] + H[2][2]

        x_prime = u/w
        y_prime = v/w

        tpts[0][i] = x_prime
        tpts[1][i] = y_prime

    #make sure transformed pts are correct dimension
    assert(tpts.shape[0]==2)
    assert(tpts.shape[1]==pts.shape[1])

    return tpts
```

3.3 Team Classification

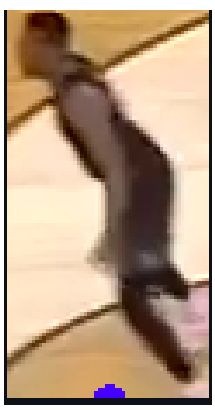
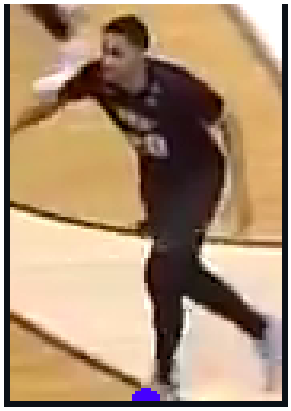
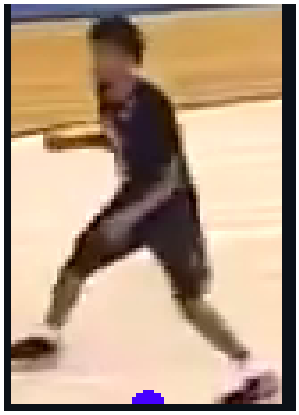
The dictionary is then passed onto another function to create separate images around all the captured players based on the coordinates of the prediction boxes along with their associated data point. During the initial development of the pipeline, a sample of pixels are collected and analyzed using a RGB evaluation function to classify the players as the home team or away team. This function is shown below. The RGB values are averaged and applied to a threshold of light and dark segmentation and passed along with the player's coordinates.

```
def create_rgb_csv():
    path = "/Users/gauravmohan/Documents/Basketball_Homography/images"
    f = []
    for filename in os.listdir(path):
        rgb_final = []
        image_file = filename
        if image_file.endswith('.png'):
            fname = image_file.split('.png')[0]
            img = Image.open(path+'/'+str(image_file))
            #cv2_imshow(img)
            fig = plt.figure(figsize=(15, 10))
            plt.axis('off')

            #print(img)
            try:
                img = img.convert('RGB')
            except:
                break
            x,y = img.size
            try:
                rgb_pixel_value1= img.getpixel((x/2,y/2))
                rgb_pixel_value2= img.getpixel((x/2+3,y/2))
                rgb_pixel_value3= img.getpixel((x/2-3,y/2))
                rgb_pixel_value4= img.getpixel((x/2,y/2+3))
                rgb_pixel_value5= img.getpixel((x/2,y/2-3))
                rgb_final.append(rgb_pixel_value1)
                rgb_final.append(rgb_pixel_value2)
                rgb_final.append(rgb_pixel_value3)
                rgb_final.append(rgb_pixel_value4)
                rgb_final.append(rgb_pixel_value5)
                f.append([fname, rgb_final])
            except:
                pass
        else:
            pass
    return f
'''
```

However, what I came to notice is that Youtube broadcast data doesn't have great resolution and moreover, when a frame is captured in the span of a player moving his body the pixels captured aren't very clean. A sample of some segmented images are shown below. This led to issues in the player

classification phase as the pixel evaluation led to inaccurate results. In order to counteract this issue, initial positions of players on the court are established in a hash table with a unique color and a function is utilized to find the next set of players.

3.4 Configuring Homography from Broadcast Angle

One major issue with using only broadcast data is that when players collectively move to another side of the court, the single camera is now moving and the dimensions of the homography change. Thus, three separate homography transformations are needed to encapsulate movement along the whole court. The program switches between the three algorithms when players are moving between the left, right, and center boundaries. The three transformations are shown below.

```
def homographyLeftTransform(im,showResult=False):
    # Calculate Homography
    h, status = cv2.findHomography(src_leftpts, dst_leftpts)
    img_out = cv2.warpPerspective(im, h, (img_dst.shape[1], img_dst.shape[0]))

    if showResult:
        cv2_imshow(img_out)
    return h
# Try out
#img_out = homographyLeftTransform(im, True)
```

```
def homographyRightTransform(im,showResult=False):
    # Calculate Homography
    h, status = cv2.findHomography(src_rightpts, dst_rightpts)
    img_out = cv2.warpPerspective(im, h, (img_dst.shape[1], img_dst.shape[0]))

    if showResult:
        cv2_imshow(img_out)
    return h
# Try out
#img_out = homographyTransform(im, True)
```

```
def homographyMidTransform(im,showResult=False):
    # Calculate Homography
    h, status = cv2.findHomography(src_midpts, dst_midpts)
    img_out = cv2.warpPerspective(im, h, (img_dst.shape[1], img_dst.shape[0]))

    if showResult:
        cv2_imshow(img_out)
    return h
# Try out
#img_out = homographyMidTransform(im, True)
```

The functions take in each frame as an image and uses OpenCV to transform the vectors of the source points onto the 2D court image shown earlier. Each function represents the set of source and destination points between the input frame and the output court image.

3.5 Additional Player Constraints

Once the images are captured along with their 2D player position, the next step is to add a couple more constraints to reduce noise from our output video. While Shapely checks if a player object is within the court, it still may pick up fans who are sitting along the sideline, because having only one camera angle makes it difficult to draw geometric boundaries around the court. Take this frame as an example to see the amount of “noise” that is picked up. The blue dots represent players that are supposedly on the court.

Once the noise is reduced from the pre-processed player coordinates, and the initial player coordinates are stored, each frame can be can fed through the data pipeline to store the set of new positions. On every third frame from the initial starting positions from the players a function is called to find the closest player to each of the individual players I have. The distances from the original positions and the set of newly collected positions are analyzed and returned. The original hash table with the player positions are replaced with the new closest position and the players are drawn onto the 2D court image along with a line connecting the initial and new positions. The temporary list of positions collected are now

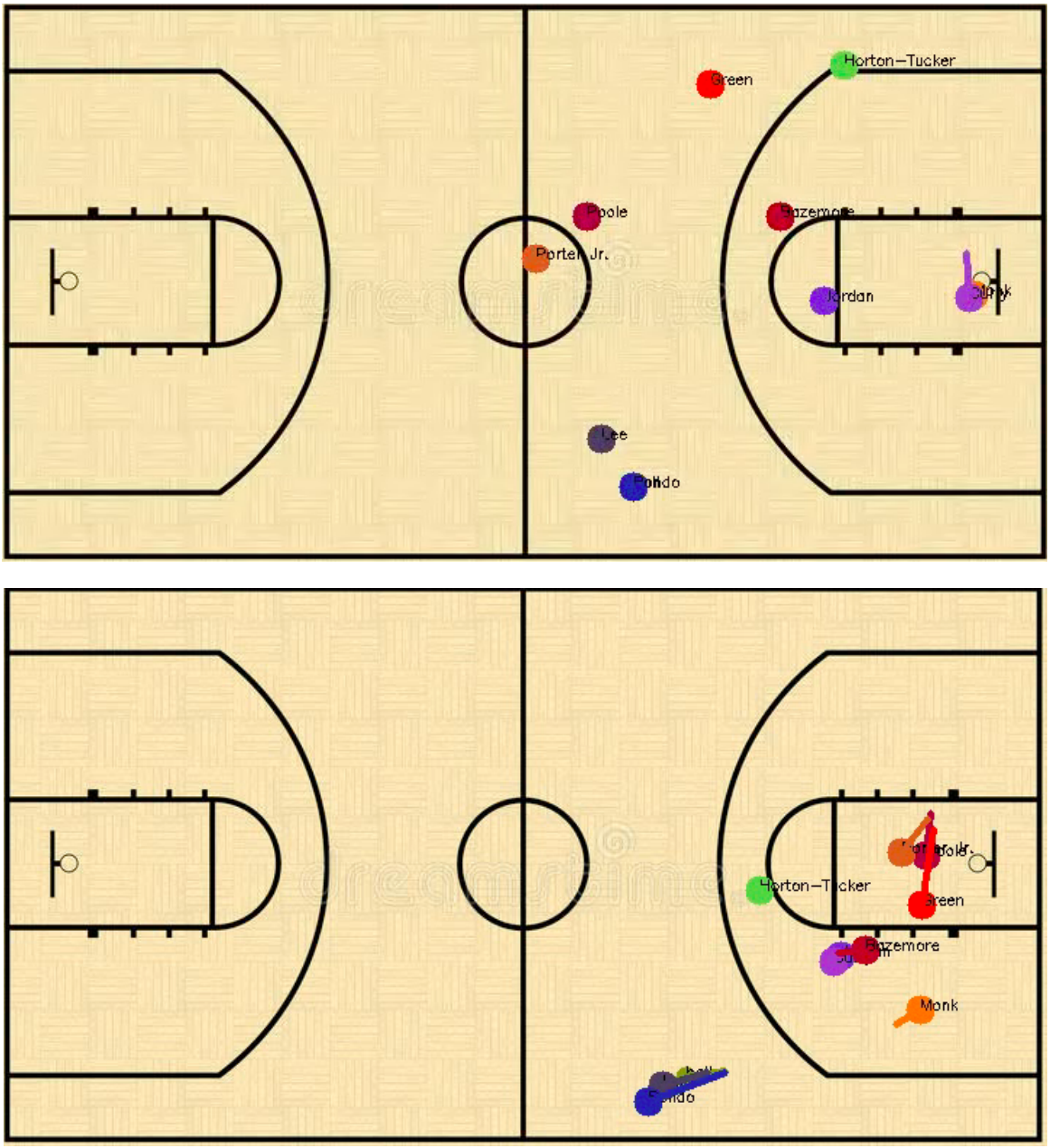
reset and will capture positions for another three frames before passing into the find_closest_player function.

```
def find_closest_player(player,positions,name):
    x1,y1 = player[0],player[1]
    cur_d = 1000
    final = None
    if name == 'ball':
        for p in positions:
            if p[1] == 1:
                final = p
        if final is None:
            for p in positions:
                x2,y2 = p[0][0],p[0][1]
                if (x1 != x2 or y1 != y2) and p[1]==0:
                    distance = ((((x2 - x1 )**2) + ((y2-y1)**2) )**0.5)
                    if distance < cur_d:
                        cur_d = distance
                        final = p
    else:
        for p in positions:
            x2,y2 = p[0][0],p[0][1]
            if (x1 != x2 or y1 != y2) and p[1]==0:
                #print(x2,y2)
                distance = ((((x2 - x1 )**2) + ((y2-y1)**2) )**0.5)
                if distance < cur_d:
                    cur_d = distance
                    final = p
    return final
```

4. Program Output and Next Steps

4.1 Frame Output

OpenCV uses a video writer to write each of these altered frames and once the video is processed fully the videowriter will close and the basketball minimap is complete. The completed minimap output is shown below. OpenCV is fairly intuitive and allows the developer to set the frames-per-second speed to increase or decrease the speed. I set the video to 6 fps, so that the player movement is very easy to see. Once each frame is processed and the next set of players coordinates are found, a function is called to draw small circles onto the screen along with the player's name or the ball. These frames are consecutively written on the screen with lines connecting the previous position to the newly found position. The following two images show the initial player positions and the new set of positions after a set of frames have been processed. Notice how the lines indicate the shift in position.



The full minimap simulation can be found on the link below:
https://www.youtube.com/watch?v=Jwx2FABfD6k

4.2 Future Improvements

There are a few ways this project could be improved. The first is to get better quality images around the captured players. Once this is implemented, the RGB pixel evaluation will yield more accurate results and classify the correct number of home and away players. This would improve the accuracy of the player

tracking, because the find closest player function can now differentiate between home and away players rather than just which player is the closest to the original player's position.

The next way to improve this project is to develop player recognition automatically by using OpenCV. If a team's roster is established, they can train the detectron library to not only detect players but to also detect their faces and match it to their database. Once again, this mandates that the quality of the image capturing provides higher resolution output.

The last aspect that could be researched further is how to detect when the video should switch between the left, right, and middle boundaries. As of right now, I am manually inputting at which frames to switch the homography transformation algorithm. But with improved detection capabilities, it could be possible to detect when most of the players are on a different boundary on the court and then automatically switch planar states. This would fully automate the pipeline and allow us to not hardcode the initial positions in. Furthermore, if a system has enough RAM they can truly simulate a whole game rather than just a snippet.

With that said, this program can allow any basketball program to create a fully functioning minimap for improved analytics tools and game film study by inputting their own raw game film footage. For more information on the full implementation of this player tracking program, please visit the GitHub attached in my LinkedIn.