Report - POVa (Computer Vision) Assignment: Ball Tracking in 3D from multiple cameras.

Dave Galea, Nikolas Papadakis, Lucas Labhini

December 2025

1 Introduction and Task Definition

The aim of this project was to create a software system capable of tracking a single coloured ball in a simple scene using at least two static cameras. The primary focus of this project is on the **3D aspects** of the problem, namely camera calibration and 3D ball localisation via triangulation.

The workflow of the project is as follows:

1. Two cameras are placed in a stable position to record the ball scene.
2. Aruco Markers are used for calibration of the cameras.
3. Two Synchronized videos are then taken of the same ball scene.
4. The two videos are then rectified.
5. The rectified videos are fed into detection code so to compute two different 2D Cartesian coordinates of the ball's centroid for each frame.
6. The 2D Cartesian coordinates of each frame are then combined into a single 3D coordinate using triangulation.
7. The 3D coordinates can be plotted and animated to see the 3D location of the ball over time.

1.1 Division of work

The work was divided as follows:

- Dave Galea: Project planning and coordination, detection, triangulation, and result visualization.
- Nikolas Papadakis: Camera setup, Camera calibration and recording of videos.
- Lucas Labhini: Rectification of recorded videos and camera setup.

2 Research and Review of Existing Solutions

This section is a summary of the research conducted on the different areas of the project, such as camera calibration, detection, and triangulation. It also includes references to existing solutions.

2.1 Camera Calibration

Camera calibration is the process of obtaining *intrinsic camera parameters*, which are the general details of the camera, such as focal length and resolution. Additionally, *extrinsic parameters*, which describe the relation and position of the cameras with respect to each other are calculated. [1]

The main existing solution used for understanding how to perform camera calibration was the github repository `https://github.com/TemugeB/python_stereo_camera_calibrate` [2], which used a checkerboard pattern and described the process of computing intrinsics, extrinsics, and how these are eventually used for calibration

2.2 Rectification

Stereo rectification involves extracting and realigning epipolar lines in a horizontal fashion so that they are aligned. This allows corresponding points in different images to appear on the same row in both images. [3].

Inspiration for our implementation and for understanding the theory and coding process of rectification was the github repository `https://github.com/xmba15/uncalibrated_stereo_rectification`[4]. Even though this solution focused on uncalibrated stereo rectification, the ideas could still be used appropriately

2.3 Detection

Detection of the ball was heavily inspired by the article 'Ball tracking in opencv'[5], which focused on detection and tracking of a single coloured ball using a single camera. This solution used HSV thresholding to isolate the specific colours which worked well since our implementation focused on a single coloured ball. Additionally, the idea to use a binary mask in our implementation also came from this article.

2.4 Triangulation

Triangulation put simply is the ability to compute depth, or, the Z-Axis from 2D images. 3D point projects onto each camera's image plane at slightly different horizontal positions, creating a disparity. Depth is inversely proportional to this disparity: points closer to the cameras have larger disparities, while distant points have smaller ones. Using the camera focal length and baseline, depth can be approximated as

$$Z = \frac{f \cdot B}{d}$$

More generally, the full 3D coordinates can be computed through triangulation using the cameras' projection matrices, providing accurate 3D coordinates. [6].

An idea of how to structure the code and get an idea of an existing implementation was obtained from `https://github.com/skimslozo/3dv`[7], which detects and tracks soccer balls.

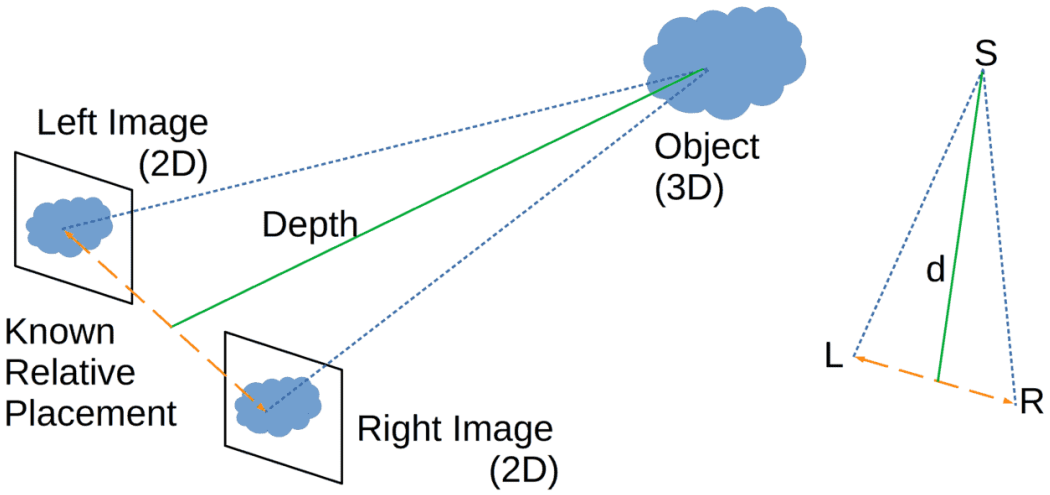

Figure 1: Simple diagram showing the concept of triangulation. Source: `https://www.baeldung.com/cs/stereo-vision-3d`

3 Implemented Solution

This section describes how we implemented our solution, which was designed in accordance to the research and techniques used in existing solutions.

3.1 Software Flow

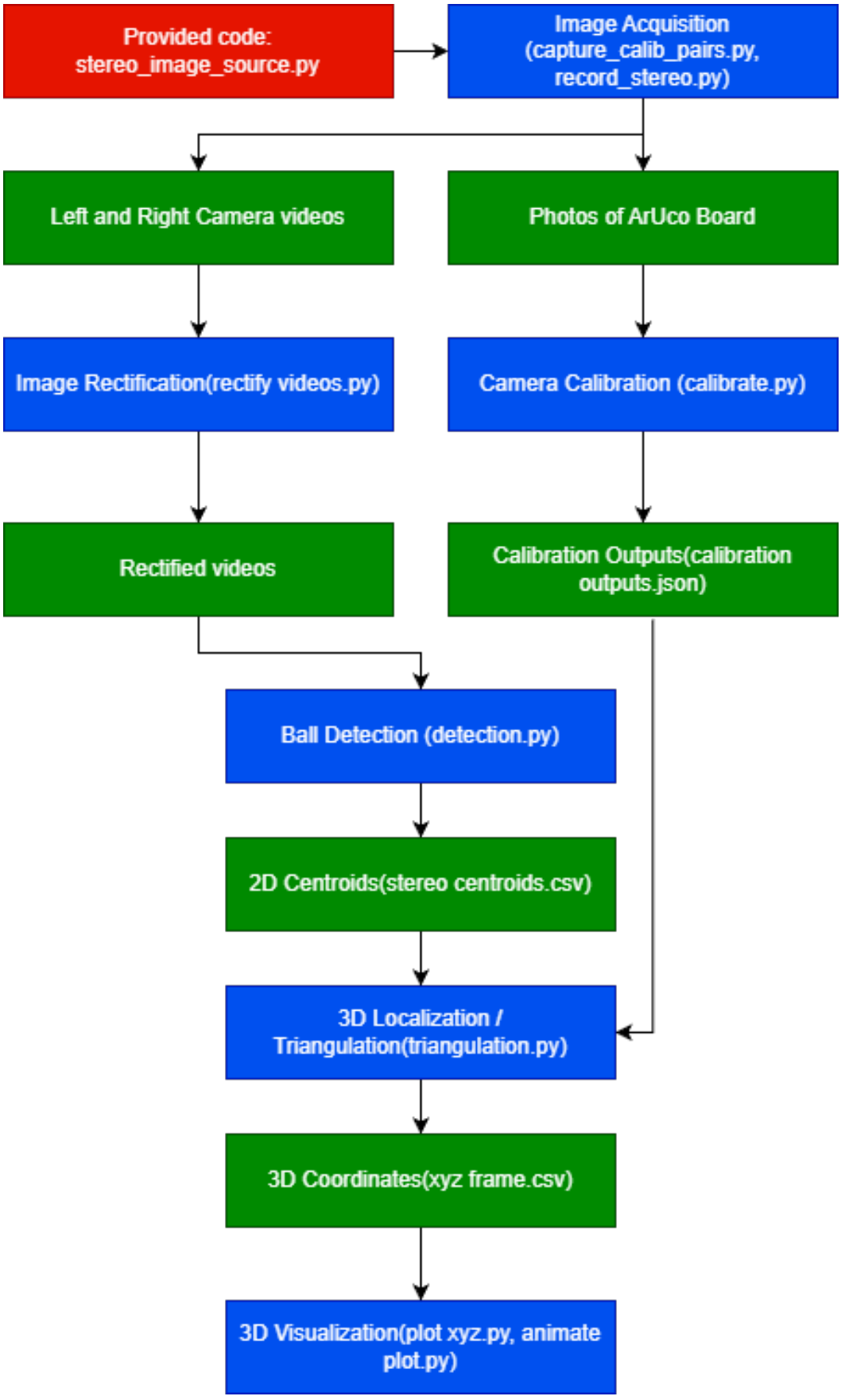

Figure 2: Software flow of the project. Red = Provided code, Blue = Code files, Green = intermediate files or outputs from code

3.2 Camera Calibration

As discussed in Section *Research and Review of Existing Solutions* (Camera Calibration), calibration aims to estimate both the *intrinsic* parameters of each camera (focal length, principal point, distortion) and the *extrinsic* relationship between the two cameras (rotation and translation). In our implementation, calibration is performed using an ArUco GridBoard, and OpenCV's ArUco and calibration routines.

3.2.1 Dataset structure and pairing

Calibration images are stored under `data/calib/<cam_id>/`, with each camera having its own folder. To form stereo pairs, our script extracts the last integer present in each filename and matches frames between the two cameras using this index. This ensures that both images correspond to the same board pose.

3.2.2 Board model and marker detection

We model the calibration target as a `cv2.aruco.GridBoard` with parameters:

- Grid size: `5x7`
- Dictionary: `DICT_4X4_50`
- Marker length: `0.020 m`
- Marker separation: `0.007 m`

For each image, ArUco markers are detected using `cv2.aruco.detectMarkers`. Since calibration requires accurate corner locations, detection is performed at full resolution (`DETECT_MAX_WIDTH = 0`). Detected marker corners are converted into matched *object points* (3D board coordinates) and *image points* (2D pixel coordinates) using `cv2.aruco.getBoardObjectAndImagePoints`. Views with fewer than `MIN_POINTS_PER_VIEW` matched points are discarded to reduce unstable solutions.

3.2.3 Intrinsics calibration and stereo extrinsics

Intrinsic calibration is computed separately for each camera with `cv2.calibrateCamera`. We use an initial guess for the camera matrix and enforce a fixed aspect ratio (`CALIB_FIX_ASPECT_RATIO`) to encourage stable focal length estimation (i.e., `fx` and `fy` remain consistent), which improves downstream rectification stability.

Stereo calibration is then performed with `cv2.stereoCalibrate` while keeping intrinsics fixed (`CALIB_FIX_INTRINSIC`). For each candidate stereo pair, we also enforce that both images contain a common subset of marker IDs, and we sort markers by ID before extracting points. This is important to ensure that corresponding 2D points from the left and right images represent the same physical board corners.

The calibration outputs are saved to `outputs/calibration_outputs.json`, including: `K1, D1, K2, D2` (intrinsics/distortion), `R, T` (stereo extrinsics), and rectification matrices `R1, R2, P1, P2, Q`.

3.2.4 Calibration results

From 25 matched image pairs, 14 stereo pairs passed our quality filters and were used for stereo calibration. The stereo reprojection error (RMS) obtained was approximately **3.81 pixels**, and the estimated baseline magnitude was **0.0676 m**. Although an RMS of 3.81 px is higher than the typical ideal target (often below 1–2 px), the resulting calibration was sufficient to proceed with rectification and 3D reconstruction, and the intrinsics produced stable focal lengths across cameras. In Section *Visualisation and Evaluation*, we further validate the practical adequacy of this calibration by comparing the reconstructed 3D trajectory with real ball motion.

3.3 Rectification

As explained in Section *Research and Review of Existing Solutions* (Rectification), stereo rectification aligns epipolar lines so that corresponding points in the two views lie on the same image row. This reduces the correspondence search to a predominantly horizontal (1D) problem and directly supports reliable stereo matching and triangulation.

Our rectification pipeline is implemented in `rectify_videos.py`. The script takes a pair of synchronized recordings (`*_color.avi`) from the two cameras and produces rectified videos stored under `data/video/<run>/rectified/`.

3.3.1 Using calibration parameters

The rectification script loads the stereo calibration parameters from `outputs/calibration_outputs.json`. In particular, it uses:

- Intrinsics and distortion: `K1, D1, K2, D2`
- Stereo extrinsics: `R, T`

Using these, rectification transforms are computed with OpenCV's `cv2.stereoRectify` using `CALIB_ZERO_DISPARITY`. We use `alpha = 1.0` to preserve a wider field of view (at the cost of potentially introducing black borders), which is beneficial for maintaining ball visibility near the image edges.

3.3.2 Rectification mapping and video output

Once the rectification transforms are obtained, a pixel-wise remapping is performed using `cv2.initUndistortRectifyMap` and `cv2.remap`. This step both undistorts the images and applies the rectifying homographies consistently frame-by-frame. The rectified left and right videos are written out for subsequent processing by the detection module.

To visually verify epipolar alignment, the script can optionally generate a debug side-by-side video (`rect\_debug\_side\_by\_side.avi`) where horizontal guide lines are overlaid on both rectified frames. After successful rectification, corresponding features across the two views should align along these horizontal lines, which is the intended property described in the research section.

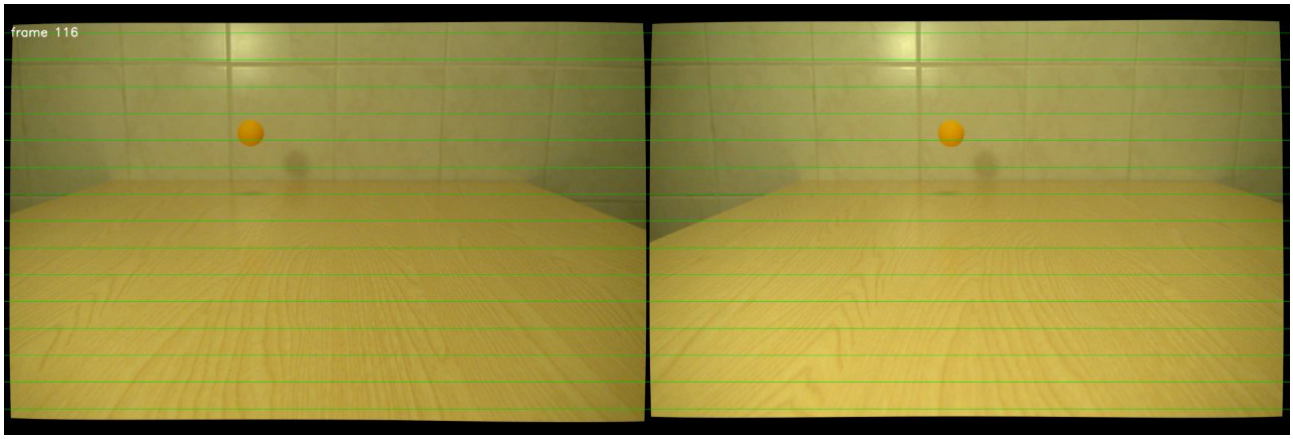

Figure 3: Rectified videos with horizontal guide.

3.4 Detection

We first convert each frame to HSV colour space and apply a binary mask using a predefined orange range:

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv, LOWER_ORANGE, UPPER_ORANGE)
mask = cv2.erode(mask, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)
```

The main hurdle with detection was certainly the presence of reflection of the ball on the shiny table surface. This was particularly difficult to handle because the reflection had very close colour ranges with respect to the ball itself; this resulted in essentially two balls being detected.

A solution around this problem was to utilize the fact that the reflection shrunk and grew proportionally with the ball, and besides for the part were the ball is bouncing, it always provides a perfect mirror image directly under it. Therefore, my idea for a solution was to increase the range for orange to be less strict to make sure that we are capturing both the ball and its reflection cleanly, then, for centroid calculation, halving the Y axis of the detected contour, and calculating the center of only the top half .

Morphological operations remove small noise in the mask, ensuring that only the ball and its reflection remain prominent. Contours are then detected in the mask, and the largest contour is assumed to correspond to the ball and its reflection.

The above can be seen below:

```
x, y, w, h = cv2.boundingRect(c)
top_half_mask = np.zeros_like(mask)
top_half_mask[y:y+h//2, x:x+w] = mask[y:y+h//2, x:x+w]
M = cv2.moments(top_half_mask)
cx = int(M["m10"] / M["m00"])
cy = int(M["m01"] / M["m00"])
```

This ensures that the centroid corresponds to the ball itself, ignoring the reflection below it. The calculated centroids for both the left and right cameras are saved in a CSV file for subsequent stereo analysis.

The result during detection can be seen below:

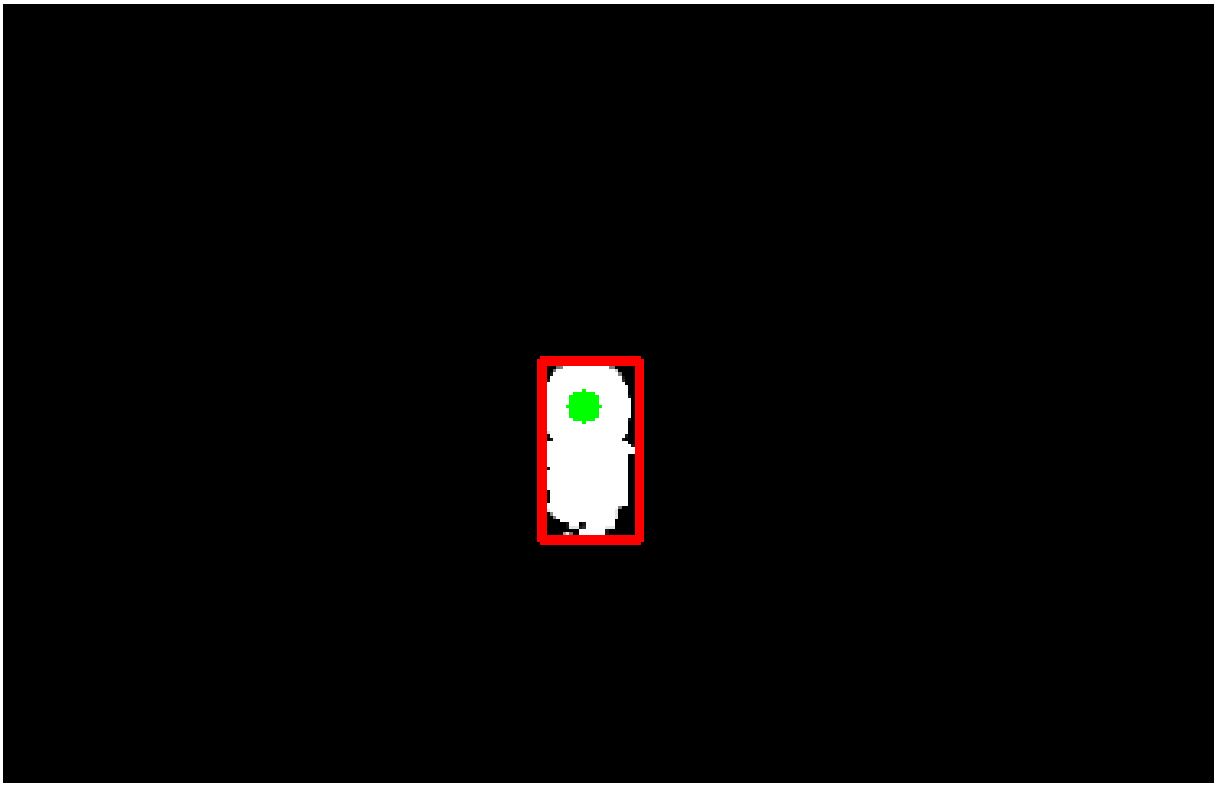

Figure 4: Detection of the ball with centroid calculatoin ignoring reflection.

3.5 Triangulation

Our implementation reads the 2D centroids of the ball detected in the left and right videos from a CSV file generated in the previous detection step. The cameras' projection matrices, obtained from a calibration JSON file, are used to perform triangulation. Points are represented in homogeneous coordinates and converted to 3D coordinates as follows:

```
ptL = np.array([[xL], [yL]], dtype=np.float32)
ptR = np.array([[xR], [yR]], dtype=np.float32)

point_4d = cv2.triangulatePoints(P1, P2, ptL, ptR)

X = point_4d[0][0] / point_4d[3][0]
Y = point_4d[1][0] / point_4d[3][0]
Z = point_4d[2][0] / point_4d[3][0]
```

The resulting 3D coordinates for each frame are then saved to a CSV file for further analysis or visualization.

One issue with this part of the project, is that for a reason we tried researching with little success, the plotting of the ball position in 3D only showed that which we saw in real life after flipping the Z axis with Z = -Z. Without this, the ball movement was mirrored from right to left (eg, if the ball was thrown from right side to left side in real life, it was from left side to right side in the visuals)

4 Visualisation and Evaluation

The way of evaluating this assignment was through visualisation of the resulting 3D coordinates following triangulation, and comparing it with the real life movement of the ball.

By comparing the following graph to the video used as for detection (video is present in `https://github.com/davgal27/3D-object-tracking/tree/main`), it can be seen that the ball can be tracked accurately both in terms of position and speed. Additionally, to make it easier to visualize, an animation from the plot is also present in the same github repository.

The graph from the plotted 3D coordinates is as follows:

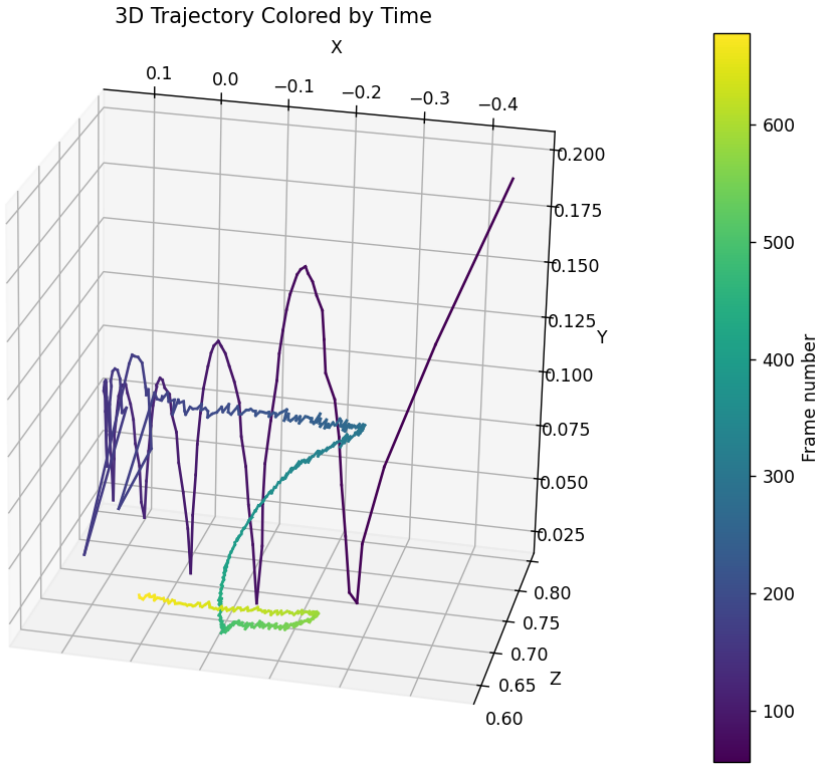

Figure 5: Plotting of 3D coordinates over time.

It can be seen that while some areas had sharp turns indicating loss of accuracy, the ball accurately depicts a bouncing motion, with each bounce losing some height, as well as accurate moving across the table when compared with the video.

5 Conclusion

During this project, we implemented software capable of tracking a single-coloured ball in space using multiple cameras. While the system successfully detects and follows the ball, the smoothness of the tracking could be improved. This limitation is likely due to calibration issues, but the core functionality of detection and tracking is working as intended.

Overall, this project proved valuable in deepening our understanding of how computer vision techniques can be extended from two-dimensional image analysis to three-dimensional spatial reconstruction. It also highlighted the potential of multi-camera systems for addressing existing vision problems and provided insight into practical challenges encountered in real-world stereo vision applications.

References

[1] B. Preim and C. Botha, Visual Computing for Medicine: Theory, Algorithms, and Applications. Amsterdam: Elsevier, 2014.

[2] T. Batpurev, "Python Stereo Camera Calibration," GitHub repository, 2020. [Online]. Available: `https://github.com/TemugeB/python_stereo_camera_calibrate` (accessed Dec. 18, 2025).

[3] Z. Chen, C. Wu, and H. T. Tsui, "A new image rectification algorithm," Pattern Recognition Letters, vol. 24, no. 1–3, pp. 251–260, Jan. 2003. doi:10.1016/s0167-8655(02)00239-8

[4] X. Ma, "Uncalibrated Stereo Rectification," GitHub repository, 2019. [Online]. Available: `https://github.com/xmba15/uncalibrated_stereo_rectification` (accessed Dec.18, 2025).

[5] A. Rosebrock, "Ball tracking with opencv," PyImageSearch, `https://pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/` (accessed Dec. 18, 2025).

[6] A. Valsaraj, A. Barik, P. V. Vishak, and K. M. Midhun, "Stereo vision system implemented on FPGA," Procedia Technology, vol. 24, pp. 1105–1112, 2016. doi:10.1016/j.protcy.2016.05.243

[7] S. Kim, "3D Vision Soccer Ball Tracking," GitHub repository, 2018. [Online]. Available: `https://github.com/skimslozo/3dv` (accessed Dec. 18 2025).