The Andrew & Erna Viterbi Faculty of Electrical Engineering Technion — Israel Institute of Technology

Electronics
Computers
Communications

VISL Vision and Image Sciences Laboratory

הטכניון - מכון טכנולוגי לישראל
הפקולטה להנדסת חשמל ע"ש אנדרו וארנה ויטרבי
המעבדה לראייה ומדעי התמונה

דו"ח סיכום פרויקט: ב'

מצלמ-שח

ChessCam

מבצעים:

אילון פררו **Aylon Feraru**

עמית בוטבול **Amit Butbul**

מנחה: **הראל ידיד** **Harel Yadid**

סמסטר רישום: אביב תשפ"ה

תאריך הגשה: אוגוסט, 2025

תודות

ברצוננו להודות להראל על הנחיית הפרויקט באופן מעולה, למדנו ממך המון ונהנינו לאורך כל הדרך.
ליוחנן על העזרה, הייעוץ, והאוזן הקשבת.
לדניאל ולאינה על המשאבים, הידע, והתמיכה שלכם.
אנחנו מעריכים את העזרה שלכם מאוד.

Acknowledgements

We'd like to thank Harel for his superb mentorship, we learned a lot from you and had fun all the way.

Johanan, for the help, advice, and attentiveness.

For Daniel and Ina for the resources, knowledge, and support.

We appreciate your help greatly.

מסמך הגדרת פרויקט

1. **חזון:**
החיבור בין שחמט למחשבים הוא תחום מרתק ומשגשג שעבר תהפוכות רבות. ב1950 פיתח אלן טיורינג את אלגוריתם השח הראשון, ב1997 גרי קספרוב, דאז אלוף העולם, הפסיד במשחק שח למחשב השח deep blue, וכיום chess.com משדרת את אליפות העולם בשחמט. שחקנים רבים משתמשים באלגוריתמי שח לנתח את המשחקים שלהם ולהציע הצעות. כדי להיעזר באלגוריתמים אלו יש לשחק דרך המחשב, או להנגיש את המשחק לתוכנה. משחקי שח רבים משוחקים עם לוחות פיזיים, והשיטות הקיימות לקבלת משוב עבור משחקים אלו מתוכנה לוקות בחסר. המערכת שלנו תחת אילוצים מוגדרים תנגיש אלגוריתמים למשחק הפיזי באופן חדשני באמצעות ראייה ממוחשבת.

2. **הגדרת משימה:**
מצלמה תשקיף על שולחן שח מנקודה מעט מוגבהת, תזהה את הלוח, החיילים והמיקום שלהם על הלוח. המשימה שלה תהיה לזהות את החיילים, התחלפות תור, ובשלב מתקדם תציע מהלכים/תאסוף סטטיסטיקות/תתעד את מהלך המשחק למען שחזור עתידי.

3. **משימות על**
   - זיהוי הרכיבים – לזהות את החיילים במשחק, סוגם (פיון, רץ, מלך...) מיקומם על הלוח, ולאיזה שחקן שייכים. עקיבה אחר החיילים לצרכי identification.
   - זיהוי מצבי משחק . (תור התחיל. תור נגמר, מצב נצחון, מצב תיקו)
   - עדכון לוח וירטואלי, ואיסוף סדר המהלכים.

NICE TO HAVE

   - להכיר אלגוריתמים מתחום השח ולבנות בעצמנו
   - לייצר פונקציונליות של chess clock
   - לתמוך בפורמטי משחק אחרים – bughouse chess , checkers, וכד׳
   - אינטגרציה עם Large Language Model כגון chatgpt שייתן הסבר על הסיבות מאחורי הצעה של מהלך
   - להנות

4. **טבלה**

משימה	זמן משוערך	מדד הצלחה
לייצר סביבת עבודה	יום	יש סביבת עבודה, כוללת את כל הרכיבים הנ״ל ומתחשבת באילוצים שהוזכרו מבוססת גיט ודוקר
לייצר דאטאסט	2 ימים	דאטאסט עם מספר רב של תמונות, מסודר בROBOFLOW
אימון רשת	3 ימים	הרשת יודעת לסווג חיילים לפי סוג, לפי שייכות לשחקן (שחור/לבן), מיקום על לוח השחמט
העלאה של הרשת לסביבת העבודה, קבלת קלט בסיסי	3 ימים	הרשת מחוברת לסביבת העבודה, יצירת ENGINE והרצה בPIPELINE עם מצלמה
בניית ארכיטקטורה	3 ימים	נבנתה מכונת מצבים, ומומשה בקוד, המערכת פועלת כמצופה (עקיבת תורים והפעלת מצבים בזמן מתאים) תחת סימולציה ורטוב
התממשקות לדאתה	3 ימים	הדאתה נשלח במקביל לאלגוריתם/מוח שח, האלגוריתם מוציא פלט של

		מהלכים ומידע, הארכיטקטורה מנתבת את הפלט למשתמש
מנהלות	יום	בניית ספר פרויקט, מצגת סוף, מצגת אמצע (כנדרש), אימון לקראת הצגה.

ChessCam — Final Project Report

Contents

List of Figures

תקציר

בפרויקט מצלמ-שח פיתחנו מערכת לניתוח בזמן אמת של לוח שחמט באמצעות מצלמת Intel RealSense D435, כדי לצלם וידאו בשידור חי של משחק שחמט, לחלץ את מצב הלוח המלא, ולהפיק המלצות מהלך בעזרת מנוע-הצעות לשחמט. בחלק סקירת המושגים הצגנו את הכלים והמדדים המרכזיים בהם השתמשנו להערכת ביצועי המודל, ובחלק סקירת הספרות סקרנו גישות קיימות לניתוח לוח שחמט אשר השפיעו על החלטות הפיתוח שלנו. המערכת משלבת טכניקות מראייה ממוחשבת, כגון מטריצת ההומוגרפיה, ורשת נוירונים קונבולוציונית YOLOv11 לזיהוי של הכלים והסוגים שלהם. בתחום מעקב האובייקטים, בחנו מספר שיטות למעקב רב-אובייקטים מבוססות מסנן קלמן: ביניהן SORT, DeepSORT ו-BoT-SORT, ובסופו של דבר בחרנו באלגוריתם SORT בזכות האיזון בין אפקטיביות לבין פשטות מימוש. המערכת שיצרנו משלבת זיהוי מדויק, לוקליזציה אמינה של משבצות, ומעקב יעיל, על מנת לספק חילוץ מצב לוח מדויק בזמן אמת. בעבודה זו אנו מרחיבים על הליך הפיתוח של מערכת זו ,מבנה המערכת, הקשיים בהם נתקלנו, כיצד פתרנו אותם, ואילו הנחות מקלות עלינו להניח על מנת שהמערכת שלנו תעבוד באופן עקבי, וכן דרכים אפשריות להתגבר על הנחות אלו. בנוסף, אנו מציגים את הישגי המערכת, שמצליחה תחת הנחות אלו לענות על היעדים ואכן לעקוב אחרי המצב של משחק שח בשידור חי וכן להפיק המלצות למהלכים מתוך מנוע-המלצות.

1 Abstract

In this project, *ChessCam*, we developed a real-time chessboard analysis system using an Intel RealSense D435 camera. Our system captures live footage of a chess game, extracts the full game state, and generates move recommendations via a chess engine. The *Term Overview* section presents key metrics and tools used for our system's creation and evaluation, while the *Related Work* section reviews existing approaches to automated chessboard analysis that informed our design choices. Our system integrates computer vision techniques such as projective transformations and YOLOv11 convolutional neural networks for semantic piece detection. For tracking, we explored multiple kalman-filter based multi-object tracking methods: SORT, DeepSORT, and BoT-SORT. We ultimately selected a slightly modified SORT algorithm for its balance between effectiveness and implementation simplicity. The resulting pipeline combines robust detection, reliable square localization, and efficient tracking to deliver accurate, real-time chess state extraction. In this work we expand on our development process, the difficulties we encountered, how we solved them, and which assumptions we had to make for our system to work reliably, as well as possible ways to overcome them. We also showcase our system's performance, and how under those assumptions the system does indeed follow along a chess game, captures the game-state, and uses a chess-engine to generate suggestions.

1.1 The Problem

According to The Guardian, "Not since American Bobby Fischer beat the Soviet grandmaster Boris Spassky at the height of the cold war has there been so much interest in the game. Chess is booming around the world" [1]. With the growing number of players looking to improve their game, the need to bridge the gap between physical games and Chess engines like Stockfish only grows. Current solutions include manual game transcription and DGT boards. Manually transcribing games is a tedious process, and error-prone. DGT (Digital Game Technology) boards provide an accurate way to record moves in real time, but they are expensive, require specialized hardware, and are not widely accessible to casual players or community clubs. Additionally, they require players to use specific pieces and maintain careful board calibration, which may not always be practical. This need can be answered by a computer vision system, which can offer accessible and immediate access for physical games.

2 Term Overview

2.1 Box Loss

Box loss [2] trains the model's predicted bounding boxes to match the actual objects in your images—or more accurately, the bounding boxes present in the dataset. The box loss is summed over object spatial locations, object shapes, and different aspect ratios, and is computed as the mean squared error (MSE) between the predicted bounding box parameters and the ground truth ones. A lower box loss means that the model is excellent at placing bounding boxes that contain the regions where objects are present. A good box-precision is required for many applications; in our case, a good box loss would help us localize the chess pieces in their correct squares more accurately. Conversely, a bad box-loss can lead to inaccurate localization and system failure.

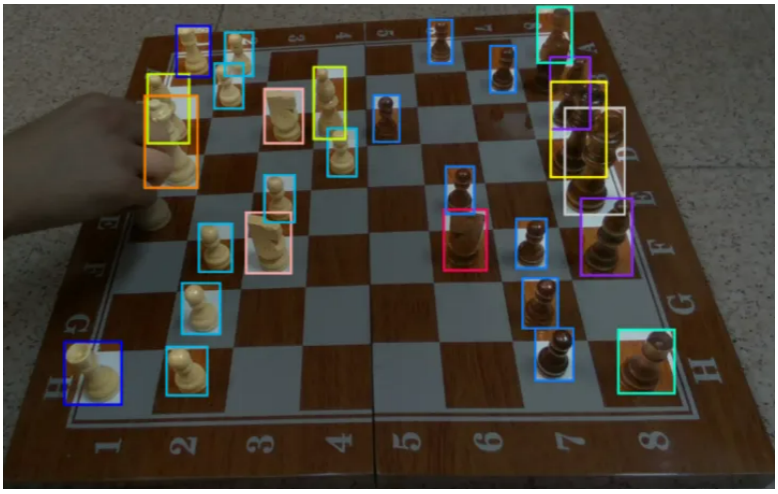


Figure 1: Bounding boxes over a dataset image

We can observe that the detections in Figure 1 are largely correct although there's an undetected black horse on the B-8 square. We'd like to train our model to recognize both the classes of the pieces present in the image, and their bounding boxes as precisely as we can.

2.2 Intersection over Union (IoU)

IoU [3] measures how well our predicted bounding box matches the ground truth bounding box. It's calculated as the area of the intersection of the predicted bounding box divided by the area of the union of the bounding

boxes. IoU is incorporated into many modern object detection architectures, including variants of Ultralytics YOLOv8 and YOLOv10, which use IoU or its variations (like Generalized IoU (GIoU), Distance-IoU (DIoU), or Complete-IoU (CIoU)) directly within their loss functions [2].

Figure 2: Intersection over Union example, taken from the Algorithms and Applications in Computer Vision Technion course.

2.3 YOLO (You Only Look Once)

YOLO is a state-of-the-art object detection algorithm that performs detection, localization, and classification in a single forward pass through a convolutional neural network. Unlike traditional two-stage detectors that first generate region proposals and then classify them, YOLO treats object detection as a single regression problem, directly predicting bounding boxes and class probabilities from full images in one evaluation. YOLO's loss function can be seen in Figure 3. This unified approach enables real-time detection speeds while maintaining competitive accuracy, making it particularly suitable for applications requiring low latency such as our chess piece detection system. The network divides the input image into an $S \times S$ grid, where each grid cell is responsible for detecting objects whose centers fall within that cell, outputting both spatial coordinates and confidence scores for detected objects. In our project we opted to use YOLOv11s, due to its ease of use as well as speed.

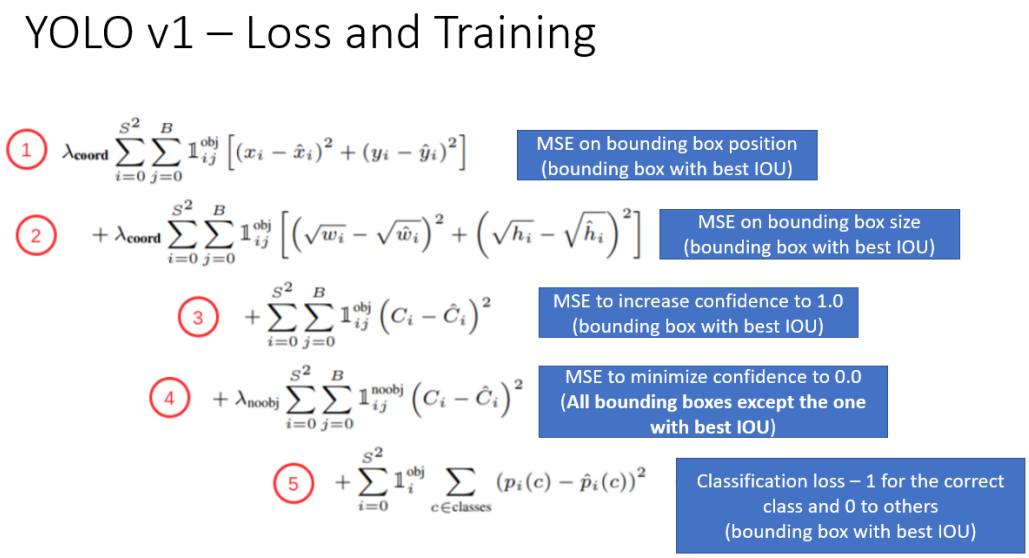


Figure 3: The loss function for YOLOv1, with component description, taken from [3]

2.4 Precision:

Precision is a parameter that tells you how precise your models predictions are. For a specific class precision is the ratio of correctly classified instances (True Positive), over all the instances *classified into* that class: Whether

Identified correctly (True Positive), or not (False Positive).

$$P = \frac{TP}{TP + FP}$$

For instance, if our model classified 5 chess pieces as pawns, where in actuality only 4 of those pieces are pawns our model has a precision score of 80% on that image for the Pawn class.

2.5 Recall:

Recall measures the ability of the model to identify all instances of a class in an image. For a specific class recall is the ratio of correctly classified instances (True Positive) over all the instances that *belong to* the class in the image, whether correctly classified, or misclassified/undetected (False Negative):

$$R = \frac{TP}{TP + FN}$$

For instance, if our aforementioned image had 10 pawns, five of which the model either misclassified, or didn't identify at all, then our model has a recall of 40% (4 out of 10 pawns correctly identified).

There's a **tradeoff between precision and recall:** - the lower the detection confidence threshold, the more prone our model is for False Positive errors and the lower our precision gets. In contrast, the model is more likely not to miss pieces since the threshold is lower so the true positive amount rises and with it recall. - the higher the detection confidence threshold, the less likely our model is to make mistakes and so False Positives decrease which increases Precision, but the model is now classifying less instances for our class and so our Recall falls. we can **Vary the threshold** for our model and plot a precision recall curve as can be seen in Figure 4.

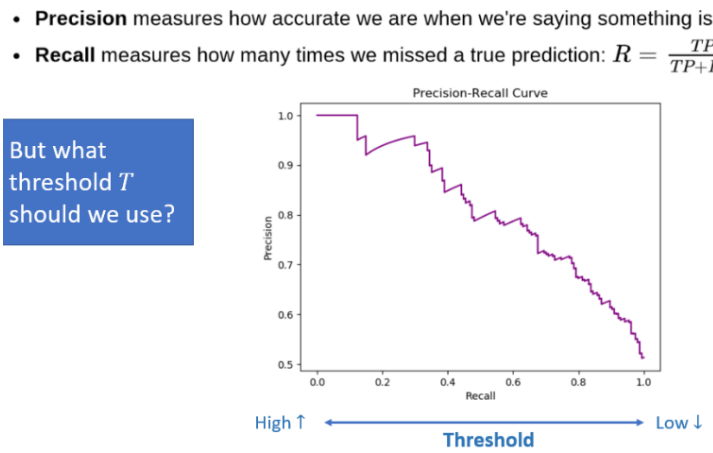


Figure 4: Precision-Recall curve with varying threshold, image taken from [3]

2.6 Average Precision (AP) and mAP@0.5:

AP computes the area under the precision-recall curve, providing a single value that encapsulates the model's precision and recall performance. an ideal value is 1. **mAP** is the mean of the average precision value across all classes. A prediction is often is often considered correct if its IoU (Intersection over Union) score passes a threshold of 0.5 (mAP@0.5), however different thresholds are also used by averaging mAP across different thresholds, for instance mAP@.5:.95. Both are automatically tracked and used to evaluate our YOLO model.

2.7 Homography Matrix

A homography matrix is a 3×3 transformation matrix that relates the coordinates of points in one plane to the coordinates of the corresponding points in another plane. It is commonly used in computer vision to perform

tasks such as image rectification, perspective correction, and planar object mapping. Given a point $\mathbf{x} = (u, v, 1)^\mathsf{T}$ in homogeneous coordinates, the transformed point $\mathbf{x}'$ is obtained as $\mathbf{x}' \sim H\mathbf{x}$, where H is the homography matrix and $\sim$ denotes equality up to a scale factor. Homographies are particularly useful for mapping points between images of a planar surface taken from different viewpoints. In our project we use homographies to map between the chess board, and a planar board that is easier to localize pieces on.

2.8 Kalman Filter

The problem our system is trying to solve is essentially an *estimation problem*: we'd like to estimate the locations of chess pieces on the board.

The Kalman Filter [4][5] is useful for solving such estimation problems, where we can tell what the actions $u_{1:t}$ and measurements $z_{1:t}$ our system performs (be it a robot traversing space, or in our case a camera that examines its own surroundings), and we'd like to use those to estimate the *state* x_t our system is in (for instance - track a piece's moving bounding box). To do that we essentially wish to compute the belief: the probability of a state, given the actions and measurements. We'd usually assume that the state with maximum likelihood state is the actual one.

$$\text{bel}(x_t) \triangleq P(x_t|z_{1:t}, u_{1:t})$$

Belief is typically hard to compute so some assumptions are made, the Markovian assumption formalizes dependence only on the most recent data:

$$P(x_t|u_t, z_{t-1}, u_{t-1}, \dots) = P(x_t|u_t, x_{t-1}) \text{ state transition probability.}$$

$$P(z_t|x_t, u_t, x_{t-1}, z_{t-1}, u_{t-1}) = P(z_t|x_t) \text{ measurement probability.}$$

The general solution to the problem, **Bayes Filter** utilizes the assumption to first estimate belief by integrating the control action u_t, and correct the estimation when new observations z_t arrive using your sensor model, but it's still quite complex mathematically and numerically intensive. If we make the following **additional assumptions**, we can get an efficient implementation of Bayes Filter - Kalman Filter.

Kalman Filter Assumptions: Linear system dynamics with Gaussian noise:

$$x_t = Ax_{t-1} + Bu_{t-1} + \mathcal{N}(0, \Sigma_u)$$

Linear observations with Gaussian noise:

$$z_t = Cx_t + \mathcal{N}(0, \Sigma_z)$$

initial belief is Gaussian:

$$\text{bel}(x_0) \sim \mathcal{N}(\mu_0, \Sigma_0)$$

and consequently, the belief itself at every point is a gaussian as well.

$$bel(x_t) = \mathcal{N}(\mu_t, \Sigma_t)$$

the Kalman filter formulation is comprised of two steps, a prediction step that based on the tracked object's assumed linear dynamics computes the location it ought to be, and an update step where based on the measurements that were recieved updates those predictions. These steps can happen in any order, and the predictions can come from multiple sensors - making Kalman filter an excellent framework for sensor fusion as well.

Prediction:

$$\bar{\mu}_{t+1} = A\mu_t + Bu_{t+1}$$
$$\bar{\Sigma}_{t+1} = A\Sigma_t A^T + R_{t+1}$$

Update:

$$K_{t+1} = \bar{\Sigma}_{t+1} C^T (C\bar{\Sigma}_{t+1} C^T + Q_{t+1})^{-1}$$
$$\mu_{t+1} = \bar{\mu}_{t+1} + K_{t+1}(z_{t+1} - C\bar{\mu}_{t+1})$$

$$\Sigma_{t+1} = (I - K_{t+1}C)\bar{\Sigma}_{t+1}$$

Where K_t is the "Kalman gain": balances the importance we attribute to measurements and the importance we attribute to motion prediction.

$\bar{\mu}_{t+1}$ is the predicted μ_{t+1} , and the $\bar{\Sigma}_{t+1}$ is the predicted covariance matrix (multi-variable standard deviation). with the assumptions we made, Kalman filter reduces the process into matrix multiplication for the expectation and covariance matrix. The bigger the covariance matrix is, the less certain we are of the prediction - the wider the gaussian distribution of $\hat{x}$

Extended Kalman Filter: The algorithm can be extended nonlinear system by linearly approximating the system at the expectation point to get new A,B,C matrices at every timestep. The idea is that while some system dynamics may not be precisely linear, we can use the first order taylor approximation to linearize them.

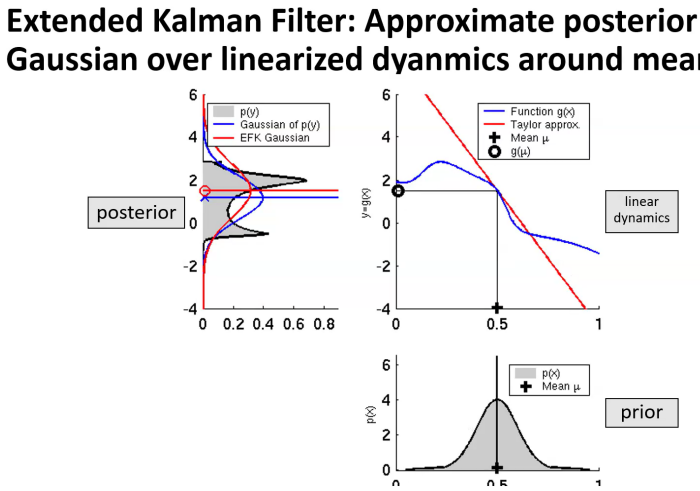


Figure 5: Extended Kalman Filter, generating posterior from prior with linearization - Mobile Robots Course Slides

we update the system based on the linearized (red) system, the blue-gaussian is the gaussian model that best fits our belief and we'd have used in a regular kalman filter whereas the red-gaussian is based on the Taylor approximation of the dynamics.

$$\bar{\mu}_{t+1} = g(\mu_t, u_{t+1})$$
$$\bar{\Sigma}_{t+1} = G_{t+1}\Sigma_t G_{t+1}^T + R_{t+1}$$
$$K_{t+1} = \bar{\Sigma}_{t+1}H_{t+1}^T \left(H_{t+1}\bar{\Sigma}_{t+1}H_{t+1}^T + Q_{t+1}\right)^{-1}$$
$$\mu_{t+1} = \bar{\mu}_{t+1} + K_{t+1}\left(z_{t+1} - h(\bar{\mu}_{t+1})\right)$$
$$\Sigma_{t+1} = \left(I - K_{t+1}H_{t+1}\right)\bar{\Sigma}_{t+1}$$

where G_t is achieved via linearization around the expectation μ_t of the approximate belief, if we take the nonlinear dynamics:

$$x_{t+1} = g(x_t, u_t) + N(0, R_t)$$

then we can linearize the noiseless system:

$$g(x_t, u_t) \approx g(u_t, \mu_t) + \frac{\partial g(\mu_t, u_t)}{\partial x_t}(x_t - \mu_t) \triangleq \underbrace{g(\mu_t, u_t)}_{const} + G_{t+1}(x_t - \mu_t)$$

We can re-write it into a similar form to Kalman Filter, with G_{t+1} taking the role of A:

$$x_{t+1} \approx g(u_t, \mu_t) + G_{t+1}(x_t - \mu_t) \triangleq G_{t+1}x_t + Bu_t$$

Note that matrix B is only used in the in the calculation of μ , which in our case, is equal to $g(u, \mu)$ so we don't really need to linearize with respect to u to get B because we already have μ.

The sensor model's observations follow the form:

$$z_{t+1} = h(x_{t+1}) + N(0, Q_{t+1})$$

$$h(x_{t+1}) \approx h(\bar{\mu}_{t+1}) + \frac{\partial h(\bar{\mu}_{t+1})}{\partial x_{t+1}}(x_{t+1} - \bar{\mu}_{t+1}) = h(\bar{\mu}_{t+1}) + H_{t+1}(x_{t+1} - \bar{\mu}_{t+1})$$

and H_t similarly takes the role of C_t, and the filter is formalized as follows:

$$\bar{\mu}_{t+1} = g(\mu_t, u_{t+1})$$

$$\bar{\Sigma}_{t+1} = G_{t+1}\Sigma_t G_{t+1}^T + R_{t+1}$$

$$K_{t+1} = \bar{\Sigma}_{t+1}H_{t+1}^T(H_{t+1}\bar{\Sigma}_{t+1}H_{t+1}^T + Q_{t+1})^{-1}$$

$$\mu_{t+1} = \bar{\mu}_{t+1} + K_{t+1}(z_{t+1} - h(\bar{\mu}_{t+1}))$$

$$\Sigma_{t+1} = (I - K_{t+1}H_{t+1})\bar{\Sigma}_{t+1}$$

Note that the linearity of the the dynamics, and the uncertainty of the prior, could both affect the quality of our prediction. Additionally, for some problems, a gaussian is a poor approximation of the belief.

2.9 SORT

SORT [6] is a multi-object tracking framework that relies on Kalman filtering in image space and frame-to-frame data association using the Hungarian algorithm. The association cost is defined by the Intersection over Union (IoU) between predicted and detected bounding boxes. Appearance information is not used—only bounding box geometry (position and scale) drives the motion model and data association. Occlusions are not explicitly modeled. The aspect ratio variable r is treated as constant, reducing state uncertainty, while the scale variable s (with its velocity $\dot{s}$) allows the tracker to adapt to size changes, helping maintain tracks through short occlusions.

In the original SORT paper, detections were provided by Faster R-CNN (a two-stage detector with a region proposal network and classification head), but the tracker itself is detector-agnostic. Each tracked object is represented by the state vector

$$\mathbf{x} = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]^T,$$

where (u, v) are the pixel coordinates of the bounding box center, s is the scale (area), r is the aspect ratio (treated as constant, so $\dot{r} = 0$), and the remaining terms are velocities. Since we track only bounding box parameters, there is no control input, and a constant-velocity model is assumed.

The Kalman filter equations take the standard linear form:

$$x_k = F_k x_{k-1} + n_k, \quad z_k = H_k x_k + v_k,$$

with $n_k \sim \mathcal{N}(0, Q_k)$ and $v_k \sim \mathcal{N}(0, R_k)$. Here, F_k is the state transition matrix, H_k the measurement matrix, and Q_k, R_k the process and measurement noise covariances. Estimation proceeds in two steps:

Prediction:

$$\hat{x}_{k|k-1} = F_k\hat{x}_{k-1}, \quad P_{k|k-1} = F_k P_{k-1}F_k^\top + Q_k.$$

Update:

$$K_k = P_{k|k-1}H_k^\top(H_k P_{k|k-1}H_k^\top + R_k)^{-1},$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H_k\hat{x}_{k|k-1}),$$

$$P_{k|k} = (I - K_k H_k)P_{k|k-1}.$$

When a detection is associated to a target, the detected bounding box is used to update the target state, with the velocity components refined through the Kalman filter. If no detection is associated, the state is propagated forward without correction until either a match is found or the track is discarded.

For a constant-velocity model with $dt = 1$, the matrices are

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

During association, each target's state is predicted and its bounding box compared to new detections. The cost matrix is built from IoU scores, and the Hungarian algorithm assigns detections to existing tracks. Assignments below an IoU threshold IoU_{min} are rejected. New detections not matched to any existing track initialize new targets (with zero initial velocity), while unmatched tracks continue to propagate without correction. Tracks that fail to accumulate enough consecutive detections are discarded to reduce false positives. Finally, a track is terminated if it remains unmatched for longer than T_{lost} frames.

2.10 DeepSORT

DeepSORT [7] takes the pragmatic approach to the MOP problem, and integrates visual data into the identification. With that addition tracking objects through longer periods of occlusions is made possible, a deep association metric is learned on a large scale person ReID (Re-Identification) dataset. SORT achieves good performance in terms of tracking precision and accuracy, but returns a relatively high number of identity switches - that is, because the employed association metric is only accurate when the state-estimation uncertainty is low. To deal with that, it's replaced by a metric that's informed by a ReID trained CNN.

The Kalman filtering framework is mostly identical to the formulation presented in the SORT paper, with the state space defined as follows:

$$\mathbf{x} = (u, v, \gamma, h, \dot{u}, \dot{v}, \dot{\gamma}, \dot{h})$$

where (u, v) are again the center coordinates of the bounding box, γ (previously r) is the aspect ratio, the height h, for each track k the number of frames since the last successful identification associated to it are counted and if it exceeds a maximum age A_{max} it's considered to have left the scene, similarly to T_{max} in the SORT paper. The probationary period before new tracks are associated to detections that aren't matched to existing objects is three frames - during which new detections must be associated to them for every frame otherwise they're discarded, like previously this is done to avoid false detections.

An assignment problem is formalized each frame to associate between detections and objects, that is solved with the Hungarian algorithm. To incorporate motion information the squared Mahalanobis distance between the Kalman prediction and measurements is used:

$$d^{(1)}(i, j) = (\mathbf{d}_j - \mathbf{y}_i)^T \mathbf{S}_i^{-1} (\mathbf{d}_j - \mathbf{y}_i)$$

Where $\mathbf{d}_j$ is the j-th bounding box, and $(\mathbf{y}_i, \mathbf{S}_i)$ is the projection of the i-th track distribution into measurement space. What this means in practice is that $\mathbf{y}_i$ is the predicted measurement for the track $\mathbf{H_k}\hat{x}_{k|k-1} = \mathbf{y_i}$ (since we need it to be in the same dimension as $\mathbf{d}$). $\mathbf{S}_i$ is the *innovation covariance* - the covariance matrix of the difference between our prediction $\mathbf{y_i}$ and z_i. It's comprised of two components, the transformed (from state-space to observation space) uncertainty of prediction, and the transformed uncertainty of measurement:

$$\mathbf{S}_i = \mathrm{CoV}(z_i - \mathbf{y}_i) = \underbrace{H_k P_{k|k-1} H_k^T}_{\text{prediction uncertainty}} + \underbrace{R_k}_{\text{measurement uncertainty}}$$

The distance metric takes into account uncertainty by measuring how many standard deviations the detection is away from the mean track location - the inverse serves as normalization, notice that it's always positive since covariance matrices are positive-semi definite. A 95% χ_4^2 confidence interval is used to reject outliers, χ^2 is chosen since the distance is the multiplication of two $d = 4$ dimensional vectors that have an assumed gaussian distribution due to the Kalman Filter model.

To account for camera motion that degrade Kalman Filter estimation quality, for each detection $\mathbf{d}_j$ a descriptor $\mathbf{r}_j$ is computed, with $\|r_j\| = 1$. A gallery of descriptor is kept $\mathcal{R}_k = \{r_k^{(i)}\}_{k=1}^{L_k}$ of the last L_k associated appearance descriptors for each object k, the second distance metric then measures the appearance distance between the i-th object and j-th detection:

$$d^{(2)}(i,j) = \min\{1 - \mathbf{r_j}^T \mathbf{r_k}^{(i)} | \mathbf{r_k}^{(i)} \in \mathcal{R}_i\}$$

This metric is essentially the minimal cosine distance between the detection's appearance embeddings and the embeddings of the i-th object's gallery, where the denominator is omitted since the vectors are normalized. The cosine distance itself is defined as

$$d(\mathbf{x}, \mathbf{y}) = 1 - \frac{\mathbf{x}^T \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [0, 1]$$

what this means in practice is that we take the dot product of the normalized descriptions in the gallery, and the description of the detection we just made. The dot product is at most 1 (when the appearance vectors are identical). A detection is very probably of some object, if that detection matches closely with at least one of its descriptors, and so we'd like the loss in that case to be minimal so we subtract their dot-product from 1. Notice that in the case of identical descriptors (for example, if we have two frames where absolutely nothing changes) $d^{(2)}(i,j) = 0$. Similarly, a CI is applied here as well.

The association problem is constructed with a weighted sum

$$c_{i,j} = \lambda d^{(1)}(i,j) + (1 - \lambda) d^{(2)}(i,j)$$

where an association is admissible if it's inside both of the confidence intervals.

Matching Cascade We'd like to avoid a global assignment problem, since there are instances where an object that's occluded for a long time grows its uncertainty, and as a result it could be favored unintuitively for far detections by the Mahalanobis distance - since it normalizes the track-distance by similarity. To avoid such cases of track fragmentations and unstable tracks, a matching cascade algorithm that gives priority to more frequently seen detections is introduced.

The matching algorithm gets a set of tracks $\mathcal{T}$ and detections $\mathcal{D}$, as well as $A_{\max}$ the maximum age. The cost matrix of admissible associations (those that fulfill the two CI requirements) is computed, and then we iterate over the track age n to solve the linear assignment problem for tracks with increasing age. In the n-th iteration, the algorithm selects tracks that have not been associated with a detection for the last n frames. For example, in the first iteration ($n = 1$), it selects tracks $\mathcal{T}_1$ that were last matched exactly one frame ago. It then solves the linear assignment problem between these tracks and the current set of unmatched detections $\mathcal{U}$, and updates the list of matches $\mathcal{M}$. This cascading approach gives priority to tracks that were seen more recently, improving robustness under occlusion and ensuring short-term continuity.

Listing 1 Matching Cascade

Input: Track indices $\mathcal{T} = \{1, \ldots, N\}$, Detection indices $\mathcal{D} = \{1, \ldots, M\}$, Maximum age $A_{\max}$

Compute cost matrix $\mathbf{C} = [c_{i,j}]$ using Eq. 5
Compute gate matrix $\mathbf{B} = [b_{i,j}]$ using Eq. 6
Initialize set of matches $\mathcal{M} \leftarrow \emptyset$
Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$
for $n \in \{1, \ldots, A_{\max}\}$ **do**
    Select tracks by age $\mathcal{T}_n \leftarrow \{i \in \mathcal{T} \mid a_i = n\}$
    $[x_{i,j}] \leftarrow$ min_cost_matching($\mathbf{C}, \mathcal{T}_n, \mathcal{U}$)
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i,j) \mid b_{i,j} \cdot x_{i,j} > 0\}$
    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\}$
end for
return $\mathcal{M}, \mathcal{U}$

Finally, unmatched tracks of age $n = 1$ (i.e., those that were last matched in the previous frame) and unconfirmed detections undergo an additional assignment step using IoU as the cost metric. This helps recover from

momentary occlusions. Note: In this step, the appearance gallery should not be updated with new feature vectors from occluded tracks, to avoid corrupting the representation.

The CNN for the appearance matching is trained offline on a ReID dataset. BoT-SORT Tracking by detection is currently the most effective method for the MOT (Multi-Object Tracking) task, where there's an object detection step and a tracking step. The tracking step is comprised of a motion model and state estimation for the bounding boxes for the objects in the following frames (typically by a Kalman Filter), and association of the new frame detections with the current set of tracks. For associations two approaches are common: 1. Localization, predominantly with IoU between the tracking model and bonding box, and 2. solving a ReID task. Both approaches are quantified into distances and used to solve the global tracking problem. Trackers exhibit a tradeoff between maintaining correct identities over time (IDF1), and detecting targets (MOTA) - IoU achieves better MOTA while ReID achieves higher IDF1

SORT based algorithms have problems. For instance, the use of Kalman Filter is sub-optimal for predicting bounding box shapes compared to the detections driven by the object detector. Additionally, KF's state characterization tries to estimate the aspect ratio γ (r in SORT) instead of the width and height leads to inaccurate size estimations. In BoTSORT the state vector is changed to address that:

$$\mathbf{x}_k = \left[x_c(k),\ y_c(k),\ w(k),\ h(k),\ \dot{x}_c(k),\ \dot{y}_c(k),\ \dot{w}(k),\ \dot{h}(k)\right]^\top$$

The observation vector changed as well:

$$\mathbf{z}_k = \left[z_{x_c}(k),\ z_{y_c}(k),\ z_w(k),\ z_h(k)\right]^\top$$

Additionally, in SORT Q and R are chosen as constants, and in DeepSORT as functions of measurements and estimations, analogizing that choice to the new state-space, BoTSORT defines Q_k and R_k as follows:

$$\begin{aligned}\mathbf{Q}_k = \operatorname{diag}\Big(&\left(\sigma_p \hat{w}_{k-1|k-1}\right)^2,\ \left(\sigma_p \hat{h}_{k-1|k-1}\right)^2, \\ &\left(\sigma_p \hat{w}_{k-1|k-1}\right)^2,\ \left(\sigma_p \hat{h}_{k-1|k-1}\right)^2, \\ &\left(\sigma_v \hat{w}_{k-1|k-1}\right)^2,\ \left(\sigma_v \hat{h}_{k-1|k-1}\right)^2, \\ &\left(\sigma_v \hat{w}_{k-1|k-1}\right)^2,\ \left(\sigma_v \hat{h}_{k-1|k-1}\right)^2 \Big)\end{aligned}$$

$$\begin{aligned}\mathbf{R}_k = \operatorname{diag}\Big(&\left(\sigma_m \hat{w}_{k|k-1}\right)^2,\ \left(\sigma_m \hat{h}_{k|k-1}\right)^2, \\ &\left(\sigma_m \hat{w}_{k|k-1}\right)^2,\ \left(\sigma_m \hat{h}_{k|k-1}\right)^2 \Big)\end{aligned}$$

σ_v and σ_m are the same constants as DeepSORT as well, dependent on frame rate. SORT-like IoU-based approaches depend on the predicted bounding box, which may fail due to complex camera motion leading to poor estimation, overlap, and match quality. Even in static positions trackers can be affected by vibrations, or wind. This is overcome via **CMC (Camera Motion Compensation):** using image registration (aligning two or more frames to get a movement model) to estimate the camera motion and correcting the KF performance with it. The implementation involves extraction of image key-points (such as corners), followed by sparse optical flow (an algorithm that estimates the motion of selected key-points between two adjacent frames of a video). The translation Matrix for the camera movement A_{k-1}^k (from time $k-1$ to time k) was solved with RANSAC. Since the background points relatively move uniformly under camera motions, and regular tracked objects can move in different ways, the use of RANSAC allows us to be more robust to their outlier nature when computing the camera motion matrix.

$$A_{k-1}^k = [M_{2\times 2}|T_{2\times 1}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$\tilde{M}_{k-1}^k = \begin{bmatrix} M & 0 & 0 & 0 \\ 0 & M & 0 & 0 \\ 0 & 0 & M & 0 \\ 0 & 0 & 0 & M \end{bmatrix}, \quad \tilde{T}_{k-1}^k = \begin{bmatrix} a_{13} \\ a_{23} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\hat{x}'_{k|k-1} = \tilde{M}^k_{k-1}\hat{x}_{k|k-1} + \tilde{T}^k_{k-1}$$

$$P'_{k|k-1} = \tilde{M}^k_{k-1}P_{k|k-1}\tilde{M}^{k^\top}_{k-1}$$

Where M is the rotation part and T the translation part of A, and A is applied on them through M and T. $\hat{x}'_{k|k-1}$ is the camera corrected version of $\hat{x}_{k|k-1}$, the predicted state vector at time k. The camera correction step is applied prior to the update step, the update step equations remain unchanged besides the usage of the camera corrected state vector and the covariance matrix.

IoU- ReID fusion An EMA (exponential moving average) is used to update the i-th object's appearance state at time k, e^k_i, based on the current matched detection f^k_i. $\alpha = 0.9$.

$$e^k_i = \alpha e^{k-1}_i + (1-\alpha)f^k_i$$

α is chosen to be high because occlusions, blurring, etc. can change an object's feature vector. Only high confidence detections are taken into account to maintain the correct feature vectors. To combine both motion and appearance information the following distance metric is constructed:

$$\hat{d}^{cos}_{i,j} = \begin{cases} 0.5 \cdot d^{cos}_{i,j} & (d^{cos}_{i,j} < \theta_{emb}) \wedge (d^{IoU}_{i,j} < \theta_{IoU}) \\ 1 & \text{otherwise} \end{cases}$$

where $d^{cos}_{i,j}$ and $d^{IoU}_{i,j}$ are the cosine and IoU distances between the object's appearance descriptor e^k_i, and the new detection f_j respectively. θ^{IoU} is a proximity threshold set to 0.5, θ_{emb} is the visual embeddings threshold set to 0.25 The result is factored in with the IoU distance to construct the cost matrix C for the assignment problem:

$$C_{i,j} = \min\{\hat{d}^{cos}_{i,j}, d^{IoU}_{i,j}\}$$

The first association step was solved with the Hungarian algorithm, and based on the cost matrix C.

2.11 Filter Choice

We selected the Kalman Filter based SORT object tracking algorithm since it combines effectiveness with simplicity of implementation as opposed to its more robust but complicated counterparts. Appearance features based ReID processes have reduced effectiveness since there are a lot of identical pieces in a Chess game. Additionally, since the motion model of an unmoving piece is relatively simple, we decided not to use an Extended Kalman Filter, and a piece that has been moved usually gets re-initialized as a separate object for having been occluded. We decided to use the BoT-SORT height-width state vector formulation as our implementation's state vector since in that article they reported better results with it, as opposed to the scale and ratio formulation.

2.12 Software Tools

2.12.1 Docker and Git

Docker is a platform for creating and running software in lightweight containers, ensuring that code runs consistently across different machines. In our project, Docker allowed us to define a reproducible environment with all dependencies (Python, Ultralytics, PyRealsense, OpenCV) packaged together. This eliminated "works on my machine" issues and simplified setup when moving between development PCs.

Git is a version control system for tracking code changes and collaborating efficiently. We used Git throughout development to manage our project repository, maintain version history of scripts and reports, and experiment with different approaches (such as different initialization methods, and detection using depth) in separate branches. This provided a reliable safety net when testing new ideas and made collaboration seamless.

2.12.2 PyRealsense

Pyrealsense is an open-source Python wrapper for Intel's RealSense SDK (Software development kit), allowing users to interact with Intel RealSense depth cameras. It provides easy access to camera streams (like depth, color, and infrared), sensor controls, and frame processing, enabling rapid development for computer vision, robotics, and 3D scanning applications. As can be seen later in the implementation part, we use it to access the color stream of our intel D435 realsense camera for processing. Additionally, we used it to access the depth stream, but eventually opted for using the color stream primarily.

2.12.3 OpenCV

OpenCV (Open Source Computer Vision Library) is a popular open-source library for real-time computer vision and image processing. It provides tools for tasks like object detection, image filtering, camera calibration, and video analysis. We use the library's pre-made functions for Homography computation, and chess-board corner detection.

2.12.4 TensorRT

TensorRT [2] is an NVIDIA developed software application for deep learning neural nets that optimizes the deep learning models like YOLO for NVIDIA GPUs. TensorRT allows the model to run at different precision formats, includes layer fusion where different layers are combined together/reduced/optimized, which reduces the computational overhead, and memory access, which improves inference speed as well. TensorRT also dynamically manages memory, and selects efficiently which functions (GPU kernels) to run parallel in a way that optimizes performance.

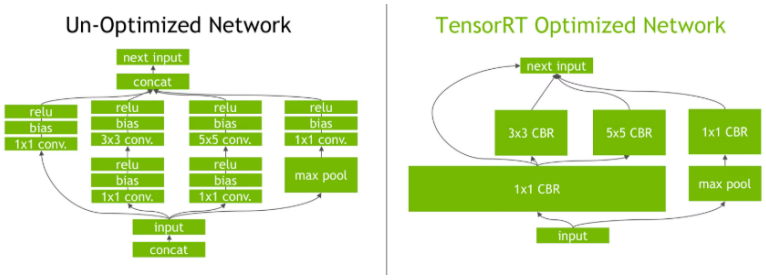


Figure 6: TensorRT layer fusion illustrated

usage is extremely simple - here's an example taken from ultralytics' website:

```
from ultralytics import YOLO # Load the YOLO11 model
model = YOLO("yolo11s.pt") # Export the model to TensorRT format, yolo11s is for small - YOLO11 has several versions including nano, small, medium, etc.
model.export(format="engine") # creates 'yolo11s.engine'
### Load the exported TensorRT model
tensorrt_model = YOLO("yolo11s.engine") # Run inference
results = tensorrt_model("https://ultralytics.com/images/bus.jpg") #run inference on bus image
```

Listing 1: TensorRT Export Example

we use tensorRT in much the same way - we export the model (this is done only when new weights become available), initialize a pyrealsense pipeline, and process the frames in the pipeline sequentially. Once that's done, we use OpenCV's `imshow()` function to display them.

2.12.5 Ultralytics and Roboflow

Ultralytics is the organization behind the YOLO (You Only Look Once) family of object detection models, providing easy-to-use implementations, pretrained weights, and training pipelines in Python. In our project, we used the Ultralytics framework to fine-tune a YOLOv11s model on our custom chess dataset, as well as to export the model into TensorRT format for optimized inference on NVIDIA GPUs. Additionaly, we used Ultralytics' implementation of a Kalman Filter: `KalmanFilterXYwH()` in our modified SORT algorithm.

Roboflow is a platform for managing computer vision datasets, including annotation, preprocessing, and augmentation. We used Roboflow to label our chess dataset, apply augmentations (such as flips, illumination changes, and mosaics), and to generate training-ready data splits (training/validation). The Ultralytics library integrated seamlessly with Roboflow's output, which simplified our training workflow.

3 Related Work

3.1 Chess Moves Detection Using Computer Vision Algorithms - Sachin's YouTube Video [8]

This approach handles live camera input using OpenCV's Python library to detect chess moves. The main steps are:

- **Chessboard extraction:** The method manually extracts the chessboard from the camera image. Although `cv2.findChessboardCorners` is commonly used, the author claims it is slow and unreliable, so an alternative or manual marking is performed to identify chessboard corners. Once corners are detected, drawing the grid lines to segment the board is straightforward. The game is captured from a top-down view.
- **Board segmentation:** The chessboard is divided into 64 distinct square images, one for each tile.
- **Move tracking algorithm:** Rather than recognizing piece types, the algorithm focuses on the occupancy state of each square—whether it is empty, occupied by a white piece, or occupied by a black piece. By knowing the initial setup and tracking occupancy changes across moves, it detects piece movements and captures. For example, if a square changes from black-occupied to white-occupied, it signals a capture. We eventually opted for a similar approach when we found the classification for class unreliable.
- **Color detection:** To robustly detect the color (white/black/no piece) of each square, a CNN classifier was trained using TensorFlow. Training data was generated by manually recording moves of a chess game, extracting the corresponding square images, and applying augmentations.
- **Pros and cons:**
  - \+ Simple and computationally light.
  - \+ Does not require piece type classification.
  - \+ Cannot start from arbitrary game positions; only works from the initial setup.
  - — Not compatible with chess variants like bughouse chess.
  - — Requires effort to create and retrain the square occupancy classifier, especially when adapting to new board types or lighting conditions.
- **Automation:** The system continuously captures frames and detects legal moves automatically, removing the need for user input after a move is made. However, camera placement is critical—top-down vertical views are ideal, while slanted views complicate detection.

3.2 Chessboard Localization and Piece Recognition from Photographs (Michael Wolz' repository) [9]

This project processes a single static image (not video) and performs chessboard localization and warping:

1. Convert the image to grayscale.
2. Apply Gaussian blur to reduce noise.
3. Detect edges using the Canny edge detector.
4. Dilate edges to thicken lines.
5. Detect horizontal and vertical lines using the Hough transform.
6. Calculate line intersections.
7. Cluster intersections to identify corner points.
8. Determine chessboard corners.
9. Warp the chessboard image to a square perspective using a projective transformation.

Key function snippet for warping using OpenCV:

```
def warp_image(img, edges):
    top_left, top_right, bottom_left, bottom_right = edges[0], edges[1], edges[2], edges
        [3]
    warp_src = np.array([top_left, top_right, bottom_right, bottom_left], dtype='float32'
        )
    side = max([
        distance_between(bottom_right, top_right),
        distance_between(top_left, bottom_left),
        distance_between(bottom_right, bottom_left),
        distance_between(top_left, top_right)
    ])
    warp_dst = np.array([[0, 0], [side - 1, 0], [side - 1, side - 1], [0, side - 1]],
        dtype='float32')
    m = cv2.getPerspectiveTransform(warp_src, warp_dst)
    return cv2.warpPerspective(img, m, (int(side), int(side)))
```

Listing 2: Chessboard Image Warping Function

This warping step corrects perspective distortion, preparing the image for further processing like slicing into 64 squares and piece classification. The function cv2.warpPerspective computes a homography matrix that maps two planes using four sets of matching points, but then it also applies the homography on the image itself, before the rest of the algorithm involves cutting it down to 64 squares and applying a model to classify each one. The data for training the neural network is based on 417 photographs of different arrangements on a single chessboard which resulted in 3900 training images (300 per chess piece + empty fields) and 7.800 validation images which is quite a lot.

3.3 Chessboard and Piece Detection Using Roboflow (James Gallagher and Shai Nisan) [10]

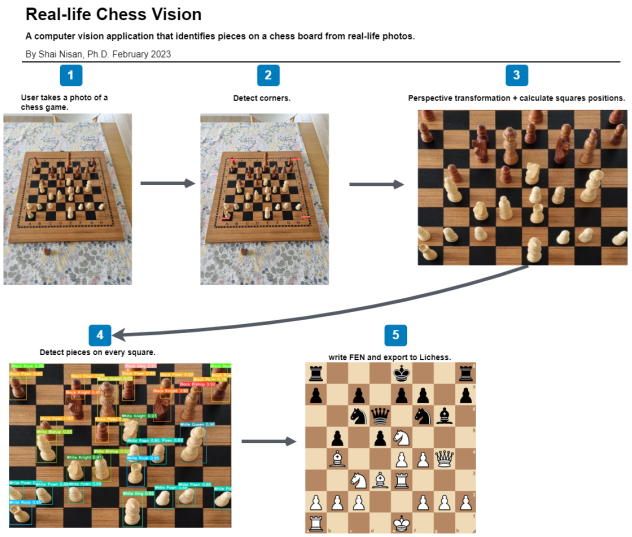


Figure 7: ChessBoard Localization and Piece Recognition from Photographs, Shai Nisan's workflow [10]

This approach breaks down the problem into:

- **Chessboard corner detection:** YOLOv8 is trained on a small dataset (50 images, augmented to 3x size) to detect chessboard corners robustly under varying lighting and orientations.
- **Corner ordering algorithm:** Once corners are detected, they are ordered reliably by computing sums and differences of (x, y) coordinates:

```python
def order_points(pts):
    rect = np.zeros((4, 2), dtype="float32")
    s = pts.sum(axis=1)
    rect[0] = pts[np.argmin(s)]     # top-left has smallest sum
    rect[2] = pts[np.argmax(s)]     # bottom-right has largest sum
    diff = np.diff(pts, axis=1)
    rect[1] = pts[np.argmin(diff)] # top-right has smallest difference
    rect[3] = pts[np.argmax(diff)] # bottom-left has largest difference
    return rect
```

Listing 3: Chessboard Corner Ordering Function

This algorithm exploits the geometric properties of points on a rectangle for consistent ordering, which is essential for perspective transforms.
Perspective transform: The image is warped to a horizontal plane using OpenCV's `cv2.warpPerspective`, but the pieces can appear distorted due to the homography assuming planarity, which does not hold perfectly for 3D pieces.
Piece detection and matching: Piece bounding boxes are detected using a trained model on the transformed images. Bounding boxes are matched to squares using Intersection over Union (IoU) calculations (e.g., via the `shapely` library). For tall pieces like queens and kings, only the lower half of the bounding box is considered to avoid overlapping detections. We'll note that the use of a homography to projective-transform the image and then either bounding boxes, or applying a CNN on the squares appears to be a repeating theme.

3.4 Chess Move Detection via Difference of Frames (Spark's YouTube Video and Repository) [11]

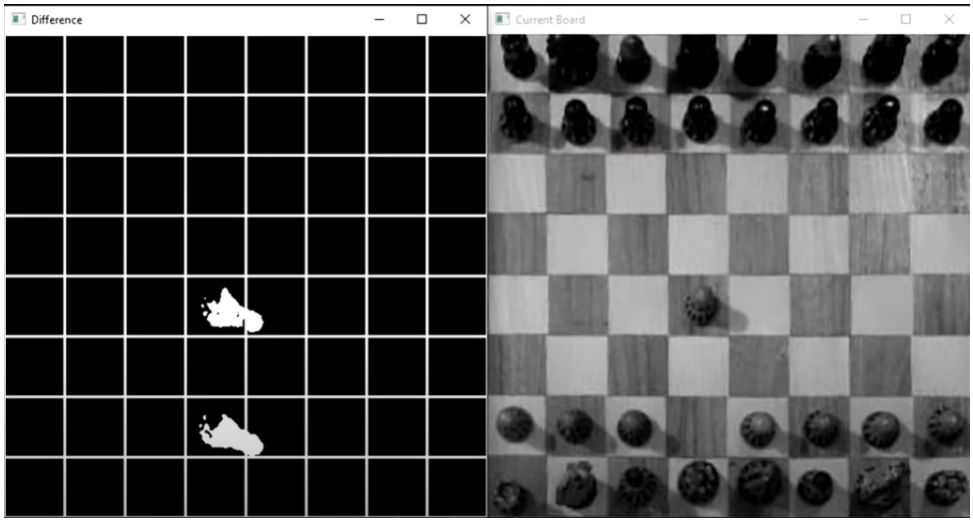


Figure 8: Difference of Frames between two game states, from [11]

The process begins with the manual selection of board corners. The image is shot from an approximately 90-degree angle and then manually cropped.
The image is then preprocessed by converting it to grayscale and applying denoising techniques. Move detection is performed by calculating pixel-wise differences between consecutive images.
To reduce noise, binary thresholding is applied. The tiles with the highest average brightness changes are identified to detect moved pieces, and the legality of moves is verified to determine their direction. A perspective transform, implemented using `cv2.getPerspectiveTransform` and `cv2.warpPerspective`, is applied in a manner similar to other works we've examined.
Castling detection is refined by examining multiple squares that show changes. Finally, user input via `cv.waitKey(0)` pauses detection until the player signals that a move is complete.

3.5 Chess Piece Detection (Craig Belshe's Capstone Project) [12]

The detection pipeline follows a structured sequence of steps. First, the system identifies the chessboard by converting the image to grayscale, applying binary thresholding, and finding contours using `cv2.findContours`. The largest contour is assumed to represent the board, though this approach may fail if other large objects are present.

Next, the corners of the board are extracted, and a homography is computed to warp the board image into a square. To avoid border artifacts, the corner positions are offset by ± 5 pixels.

Grid locations are then determined either manually or through a multi-step automated process that includes heavy blurring, Canny edge detection, hysteresis thresholding, and Hough line transforms, which help to handle slight misalignments.

Once the grid is established, chess piece detection is performed on the warped images using YOLOv4 for classification. The bottom center of each detected piece's bounding box is then transformed into board coordinates to map it to its corresponding square. This bottom-center mapping is effective even when pieces overlap due to perspective. In our own project we borrowed this technique but added a slight vertical offset since some bounding boxes exceed square boundaries.

Finally, the system's performance was evaluated on 20 web-sourced images. The approach proved reliable in most cases but struggled with images where the board was far from the camera and the background caused false detections.

3.6 Literature Report Conclusions

We can notice that the workflows are particularly varied, however, all of them either film the chessboard from a directed above angle, or apply a projective/Homographic transformation on the image.

Piece detection is most commonly performed using convolutional neural networks (CNNs) object detection models such as YOLO, trained either on entire board images or on cropped square regions. While direct per-square classification (as in [8]) simplifies the task by avoiding piece-type recognition, it imposes constraints such as requiring the game to start from the initial setup and retraining for different boards or lighting. In contrast, full object detection approaches ([12], [10]) offer greater flexibility in recognizing arbitrary game states but demand larger, more diverse datasets and careful handling of occlusions and distortions.

Board localization methods also vary considerably—from purely manual marking of corners to automated techniques leveraging Hough transforms, Canny edges, or deep learning-based keypoint detection. Regardless of method, most pipelines converge on performing a projective transformation to normalize the board's geometry before further processing, or a simple crop if the camera is already at a 90 degree angle.

A recurring challenge across all reviewed works is robustness to non-ideal conditions: skewed camera angles, varying illumination, piece occlusion, and background clutter all degrade detection accuracy. Some approaches mitigate these issues via aggressive data augmentation, while others rely on strict physical constraints such as fixed camera positions and controlled lighting.

We decided to use the same projective transformation techniques, and `opencv`'s corner-detection to detect the required corners as matching keypoints for the Homography computation. We also decided to use YOLOv11s (a CNN) for the detection, due to the network's balance of size, performance, and ease of use in the Ultralytics package. We used the same method of projecting points as Craig Belshe [12] had, and when we found our own detections were unreliable we used the same move tracking algorithm Sachin implemented.[8]. We also sorted our corners similarly to Shai Nisan [10] to make the found corners match correctly to the chess pieces.

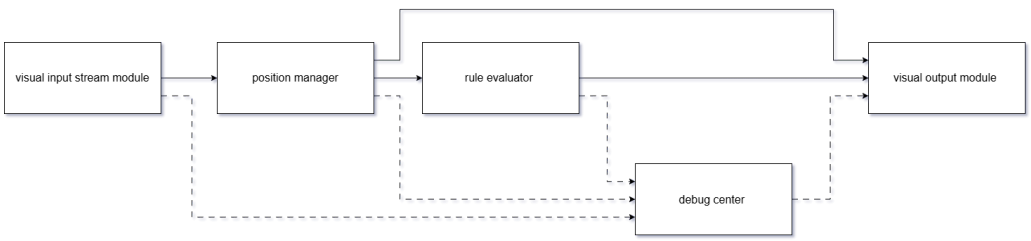
4 Method

4.1 Overview

4.1.1 Planning

We decided to create a detailed plan before starting active development, and in retrospect we can see that the in-depth planning benefited the project immensely. We decided to use the Docker platform so that we can develop consistently without having package problems. We decided to develop in python as opposed to C++ since Ultralytics, and Realsense offer support and documentation in that language, and we're familiar with it. We started out by dividing the task to several primary modules, each with their own responsibilities, and then we implemented all of them.

4.1.2 System Module Diagram



Video Stream Module:

- Sends camera stream to the output model for visualization.
- Sends processed detection results from the camera stream to the Position Manager.
- Sends corner detection data to the position manager.
- Sends detection errors to the Debug Center.

Position Manager Module:

- Sends piece and board detection/localization errors to the Debug Center.
- Sends piece and board localization data to the rule evaluator.
- Can connect to the Visual Output module to display the virtual game state representation when required.

Rule Evaluator Module:

- Constructs game state, evaluates the legality of the move, and fetches move suggestion data.
- Sends suggested moves for visualization to the Visual Output module.
- Sends reports of faulty move suggestions or invalid game states to the Debug Center.

4.1.3 visual input stream pipeline:

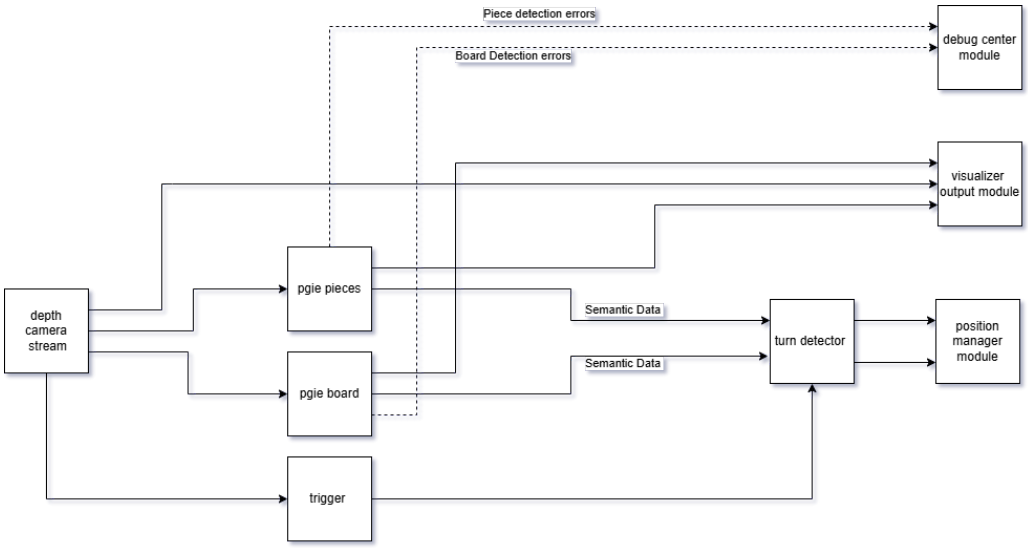


Figure 9: The visual input stream pipeline diagram

Trigger is responsible for indicating whether or not a turn has been completed (if a piece move has been completed).
The inference engines `pgie pieces` and `pgie board` are responsible for identifying the piece types and locating the coordinates for the board's corners respectively.
Output visualizer module receives raw camera stream, as well as semantic detection data (bounding boxes and piece locations), and visualizes them.
Position manager module receives semantic piece and board detection data.

4.1.4 Position Manager:

The position manager module is responsible for computing the homography using the corner data, managing the tracked objects, deleting and creating them. Additionally when a move input is pressed, the manager module homographically transforms the pieces and localizes them into squares, passing that data to the rule evaluator.

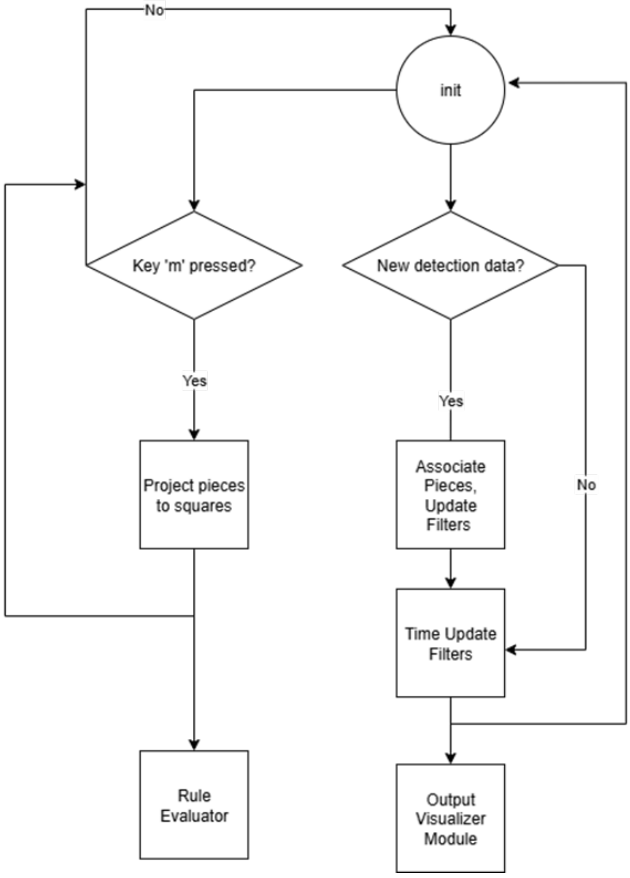


Figure 10: The position manager logical process diagram

4.1.5 Data Visualizer

The data visualizer outputs to screen both a video output) and a textual output — chess-engine-generated move suggestions, into the program terminal. The data visualizer also presents the bounding boxes, as well as the square representing points that are projected to localize piece squares.

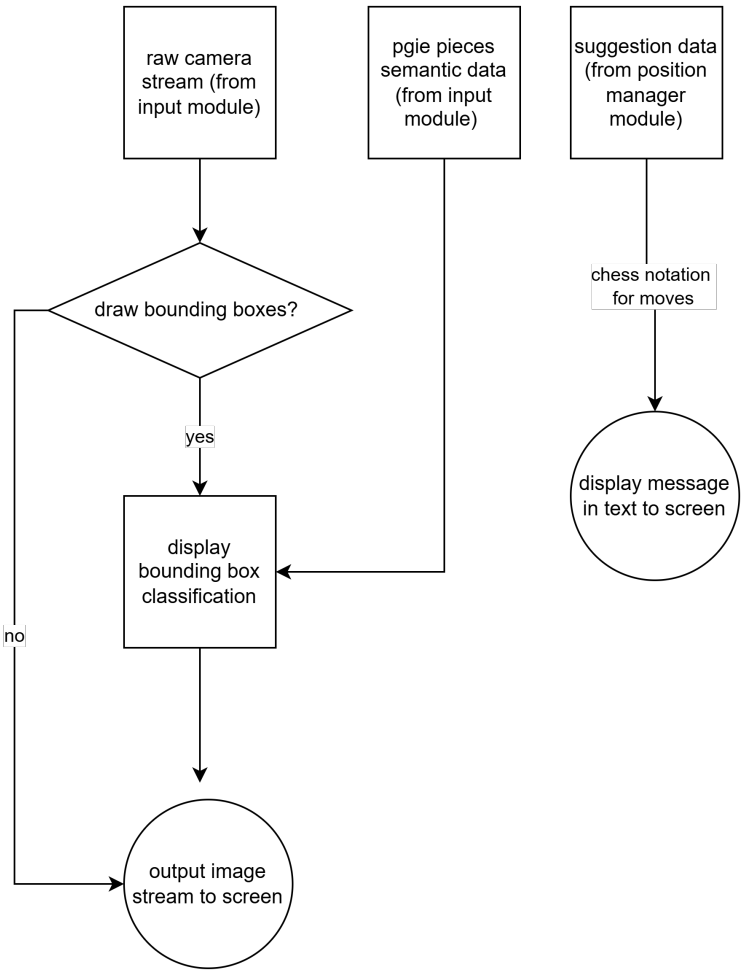


Figure 11: The data visualizer logical process diagram

4.2 Implementation Details:

4.2.1 System UML Diagram

Our system was programmed using Object oriented design, wherein we define objects and classes such that their instances, methods, fields, attributes and interactions help define our system. We built a UML diagram to illustrate our system Classes, and we evolved it as our project progressed. For instance, we added the PieceTracker object, and made the GameState object more robust as we realized we couldn't consistently rely on YOLO for correct piece classifications. The UML diagram can be seen in Figure 12. The objects are expanded upon in the relevant sections.

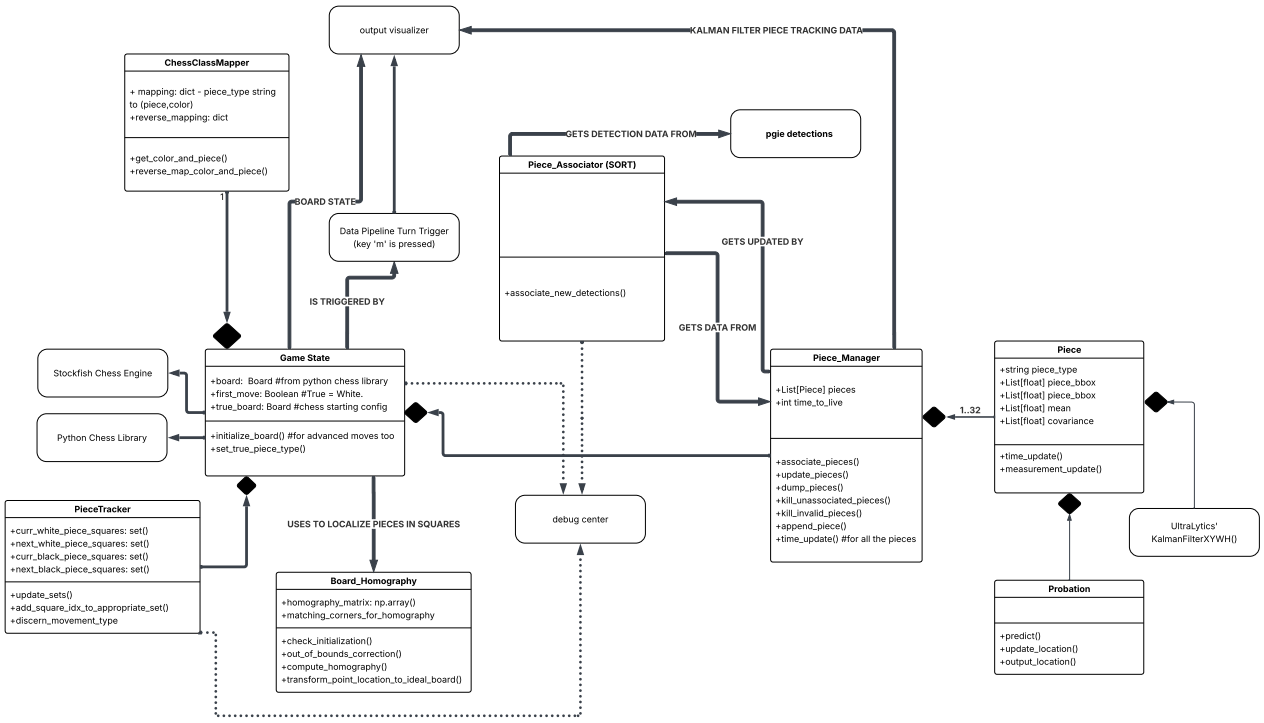


Figure 12: Our system UML diagram, we can see how the piece manager, the piece class, and others interact.

4.2.2 Piece Detection Dataset Creation

For a successful classification of the pieces we wanted to tune our YOLO model, our model should ideally handle the detection consistently during a game with scenarios such as disturbances, on a variety of game positions. To facilitate that we decided to use the Roboflow platform that streamlines model generation and training-data tagging.

Initially, we used a chess detection dataset made by Roboflow, with chess pieces labeled in English containing 292 samples over a green-chessboard in an above position similar to the one we decided to adopt for our dataset.

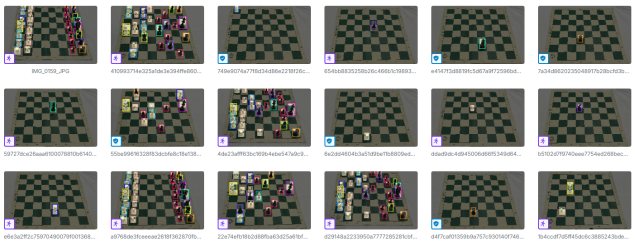


Figure 13: Some of the images contained in the original Roboflow dataset

We forked the dataset and added our own 60 images that we have taken using the intel camera mounted on the tripod to help the model generalize to our different chess-set, camera and pieces. We filmed at the highest resolution possible with our camera 1280×720 since the more details we pass onto the model the better it can distinguish between different pieces.

Figure 14: our camera setup, an intel realsense depth camera mounted on a tripod, overlooking our chessboard.

We created a script called `recorder.py` for easy recording with the depth camera that starts and stops at the press of a button, displays both the RGB and depth stream, and allows for multiple recordings using the pyrealsense library - since we knew we'd need to make use of the script a lot, the design principle we followed was to invest time into making it as easy to use as possible. We recorded both random game configurations and some famous chess-game position to make sure we had a variety of different sources to choose images from and a diverse dataset.

We followed a clear design principle: maintaining a consistent file hierarchy throughout the project. Scripts, datasets, and other resources were organized into folders according to their purpose, ensuring a clean and navigable workspace.

The Intel RealSense camera outputs data in a format called `.bag`. To process this, we used the RealSense library to implement `bag_to_img.py`, a script that opens these recordings as streams and extracts color frames at regular intervals. From these frames, we selected key images, which were then uploaded to Roboflow for annotation.

The file `recorder.py` was placed in the `scripts/` folder, while `bag_to_img.py` was located in the `scripts/utils/` subdirectory.

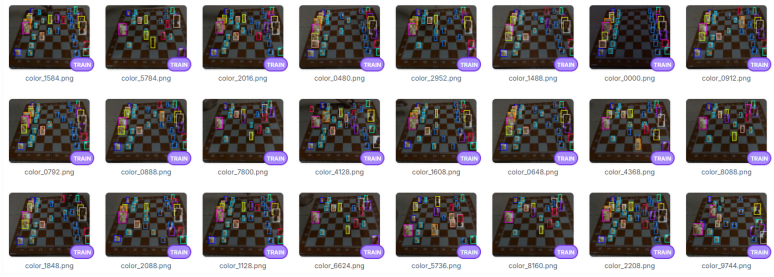


Figure 15: some of the images we tagged on our own, notice how the angle is slightly different, and the chessboard is as well.

We also made sure to make about a third of our images contain occlusions such as hands, a tilted board, and null images (with no pieces), since those scenarios could be present as well in the game and our model needs to learn how to account for them, as can be seen in Figure 16.

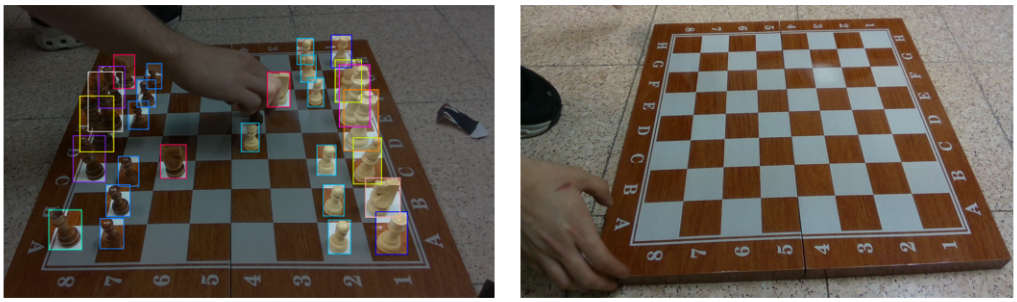


Figure 16: an occluded image and a null image from our dataset.

4.2.3 Pre-processing and Processing the Dataset

Training Validation and Test set splits are a well known procedure that many deep-learning datasets undergo where the dataset is split three-ways, the majority of the data is used to train the model (training-set) via some gradient-descent optimization scheme (commonly ADAM or ADAMW), during training the current model's weights are checked against a validation set to determine whether or not the model overfits, or generalizes well to the data it did not train on (during the validation process weights aren't updated), and at the end of the process the model's performance is tested against a test-set.

In our case and many industry cases, the existence of a test set is not strictly necessary since we test our model's performance *during deployment*, so instead of allocating our samples to a test set we'll allocate them to either training or validation - making more from our data.

A good rule of thumb is to allocate 10-20% of the images to validation in a large dataset and 20-30% of the images to validation in a small dataset, the more we allocate to validation the less we can train on but the more varied our validation set will be and the more robust our check against overfit. Our dataset contains 348 images and we elected to allocate randomly 67 of them to the validation set $\sim 20\%$

once our dataset images have been assembled we need to pre-process and augment them, which we partly selected deliberately in Roboflow and partly were applied natively by the ultralytics python training algorithm. The augmentations we selected are stretching the images to be square 1280×1280 images, mosaics (cutting and combining the images together), flips, and distortions that mimic the camera such as $\pm 10\%$ illumination variance. Since we didn't add many augmentations our image count was still relatively low so we duplicated the images in the dataset.

The training resulted in a mAP@50 score of 98.8%, Precision of 98.8%, and Recall of 99.4%, these values are very encouraging and so far the model appears to train well.

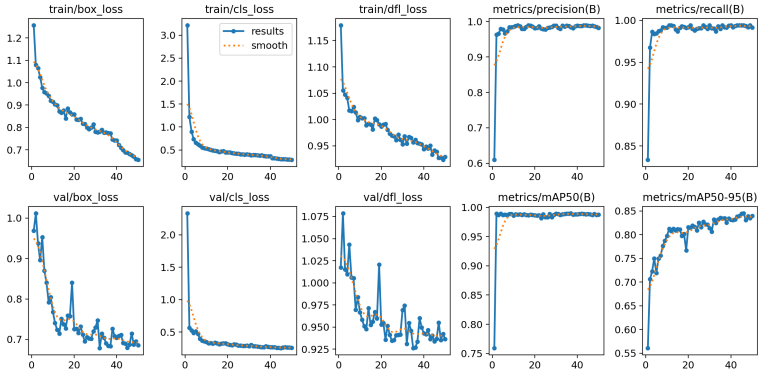


Figure 17: Training metrics/loss function graphs and tracking how they update during training. Details on losses present in the Term Overview section

we'll notice how a smooth and steady decline in both training and validation loss reinforces our assumption of a well trained and generalized model, an assumption we'll test in deployment. Now that we've acquired our weights we'll use the TensorRT export format to optimize the model for real-time inference on NVIDIA GPUs.

4.2.4 Visual Input Stream Module Implementation

We decided to base our model on the RGB functionalities of the camera for a start since YOLOv11s is generally trained on three-channel inputs, and optionally expand to RGB capabilities when the model succeeds. To create the data streams we used `pyrealsense` to establish a frame pipeline, we defined it here:

```
pipeline = rs.pipeline()
config = rs.config()
config.enable_stream(rs.stream.color, 1280, 720, rs.format.bgr8, 30)
pipeline.start(config)
```

Listing 4: pyrealsense frame pipeline definition

We used the highest resolution available for our intel D-435 camera, to give as detailed data as possible for processing.
And inside a program loop that repeats every frame we fetch the frames and feed them to the inference engines. The visual input stream module is implemented using the trained YOLOv11s model as the `pgie pieces` block. This block outputs a stream of semantic data in the form of bounding box objects, where each object contains the bounding box center coordinates, height, width, and label for each frame.

Most of the visual module's implementation is available in the `online_inference.py` file.

Figure 18: Drawn corners found by Opencv's function, acting as a secondary inference engine. This is the required corner orientation for the inference engine to behave properly.

4.2.5 Position Manager Module Implementation

square-localization
The position manager was a more involved module, comprised of several objects, it served as the main data processing unit.
We decided instead of projecting the entire image, to represent the detection of a piece as a single point and then project it with the homographic matrix over to an "ideal board" that makes square localization easy. We opted for that unlike a lot of other projects that project the entire image so that we can avoid a big computational step during inference. To do that we decided that a point in the lower quarter of the bounding box (in terms of height, $y_{pt} = 0.25h + y$), and in the middle ($x_{pt} = x$ in terms of width) will represent a piece's square well. You can see that illustrated in Figure 19, where the representation point lands on the square of the piece.

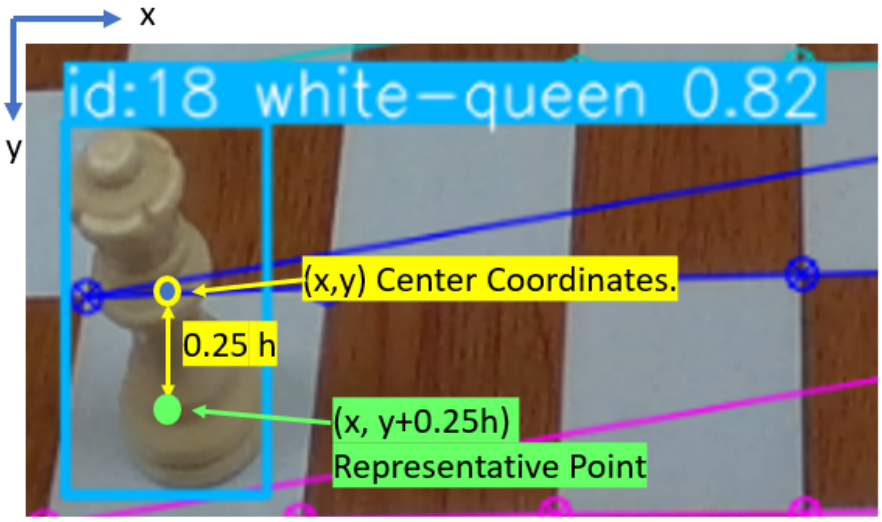


Figure 19: How a point representation for a piece is computed based on bounding box parameters (x, y, w, h). Notice how the green point better illustrates the square a piece is on compared to the center point.

We took the detections and associated them with a squared-distance based SORT Associator object to Piece objects, and propagated that data to a Game State object to represent the board.

The piece object had fields for Piece type, and it contained an instance of Ultralytics' Kalman Filter for tracking with state $(x, y, w, h, \dot{x}, \dot{y}, \dot{w}, \dot{h})$. We opted for this Kalman filter since in [13] Aharon et al. claims this state vector yields better results in their experimentation than using aspect ratio. We built a Piece Manager object to hold all the pieces, discard ones that got no detections, and ones that were out of bounds.

To find the location of the pieces on the correct squares we built a Homography object to compute and return the transformation of points that represent piece locations between the board. To get the points for the matrix computation we computed the corners in the image with CV2 and matched them to the appropriate corners in the ideal chessboard.
We computed the set of representative points for the Piece Objects, and used that for localization.

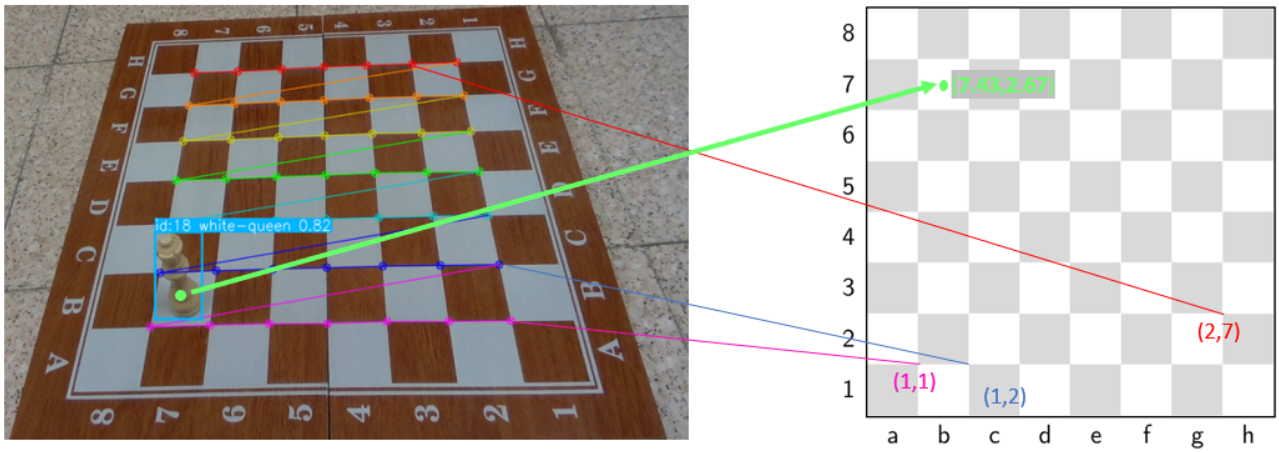


Figure 20: Homography matching point and piece localization illustrated. We use all 49 corresponding point pairs, but the schematic shows only three for clarity.

We'll note that once a piece is on the transformed board, localization becomes simple, since by design warped

coordinates are in direct relation to the corners. For instance, a point at $[7.43, 2.67]$ like in Figure 20 is on the $(7, B)$ square.

Piece Management

Since we have more than a single piece, we now require logic to attribute detections either to existing pieces or new pieces, as well as logic to discard pieces that receive no detection. To do that we created an Associator object that the piece manager calls on to solve the Linear sum assignment for Detections. We decided to use a squared distance metric between the 'square-representing' points of the detections and Piece location estimations.

Additionally, we attached to each Piece an object called Probation, that places new tracked objects under a probationary period. When creating a new track, the tracklet has to receive associations every frame for 5 frames, otherwise the manager would discard it.

Another Filtration method we added was time-to-live, where if an object does not receive a detection association for 5 frames, the object is discarded. Both of these ideas are taken from the SORT paper [6].

The Associator is called every frame, and every frame each Piece's Kalman Filter is time updated, and if a Piece object receives a detection attributed to it, they get updated via the measurement. In contrast, the Board object held inside the GameState object that is responsible for consolidating the detections into a stockfish-readable format, is updated only once the move trigger indicates a move has been completed.

5 Experiment

5.1 Mid-development Results

Our system seemed on the whole to detect pieces with reasonable accuracy, but it was far from perfect. We noticed that square localization, after adjusting and fixing corner order to match that of our corner array in the idealized board, performed well consistently. But piece identification faced several issues. The first of those were erroneous classifications: our model was certain that one piece was another despite training it. An example can be seen in Figure 21.

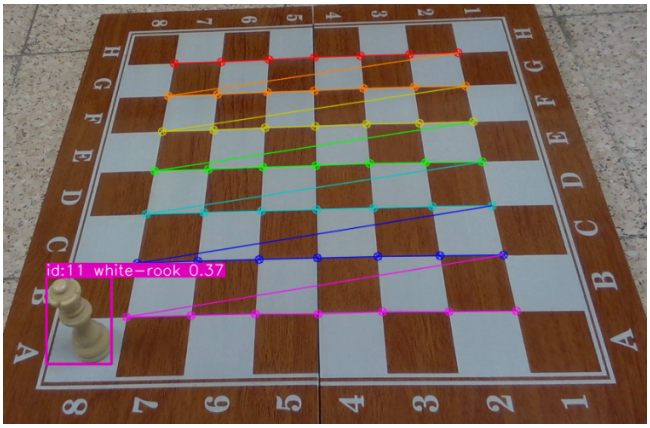


Figure 21: Misclassification example - the model is reasonably certain a white queen is a white rook

Another error we encountered quite frequently is label-flicker, wherein the label of a piece the model is unsure about would change in a frame-by-frame basis. An example can be seen in Figure 22.

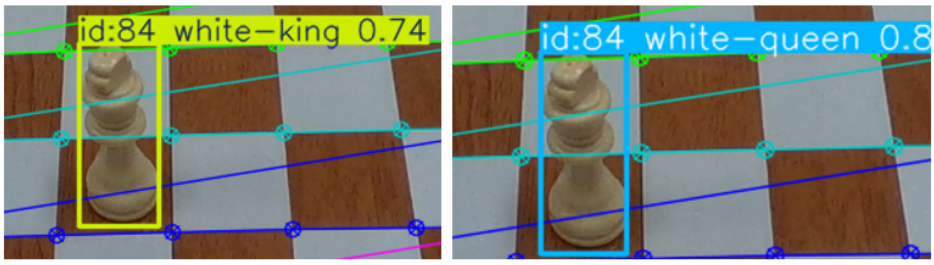


Figure 22: Label Ficker - label changes on a frame-by-frame basis. Images taken a frame apart.

Additionally, we falsely assumed that people would remove taken pieces out of the camera frame - where in game by habit taken pieces are sometimes placed in view of the camera, which can cause them to be falsely attached to a board square or cause an out of bounds error. Another instance of an out of bounds error is caused by misclassifications of everyday objects as pieces. An example of both instances can be seen in Figure 23, where the bishop can be mis-attributed to a square on the board/cause an out of bounds error, as well as the chair wheel.

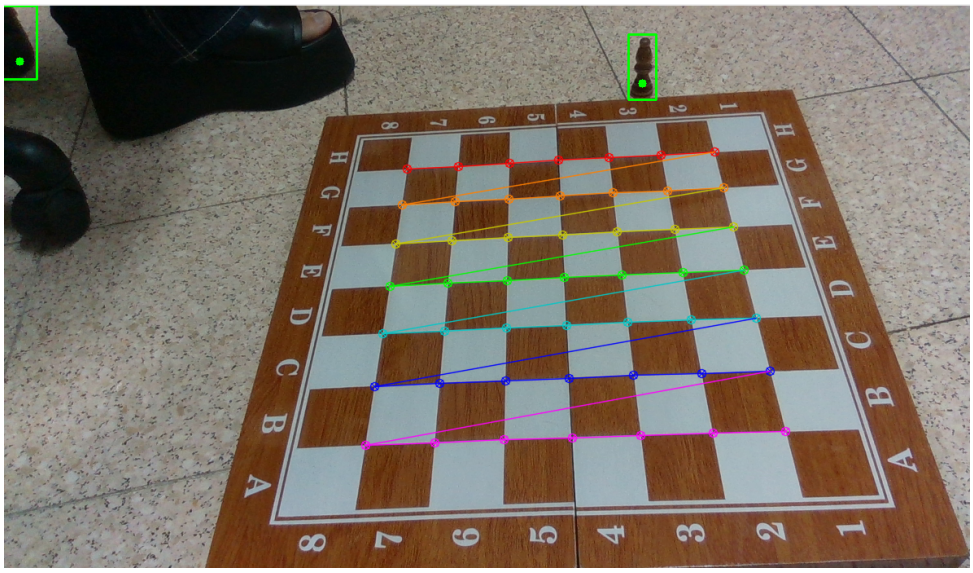


Figure 23: Problematic out of bounds detections

Association also seemed to work well for the most part, but for some adjacent pieces there's been a frequent addition of a false object between the two. An example can be seen between the two left pawns in Figure 25

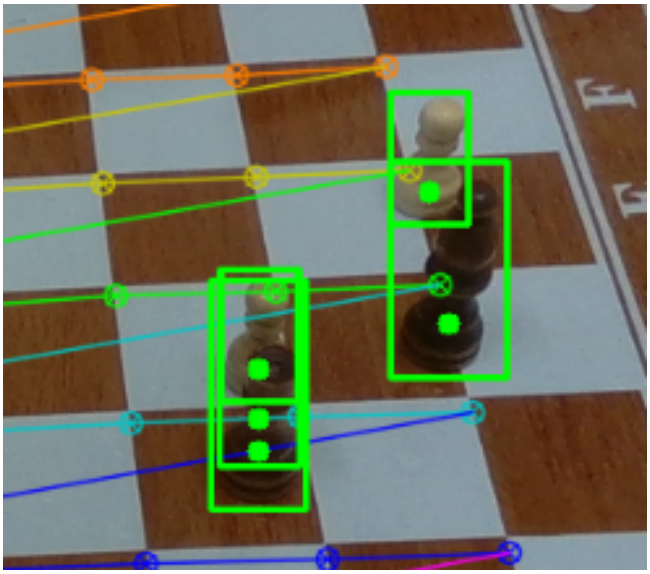


Figure 24: Phantom Object Error

The most severe error was the misidentification of pieces since we directly relied on consistent identification for the system to work properly. To solve the out of bounds issue, we decided to give a slight tolerance and include objects that their transformed square representation points exceed the board limits only slightly, and reject those that exceed it majorly. The threshold was found via experimentation. To solve the phantom object error, as well as other errors related to Kalman filter behavior going awry, we added the probation logic and the time-to-live logic, which seemed to lessen the severity of the problem but not solve it entirely. But we needed to add significant design alterations to deal with the misidentification error, those changes and their ideation are discussed in the next section.

5.2 Design Improvement Considerations

We noted that the biggest problem we had to solve was inconsistent detections. We decided to consider several different approaches, and then select ones that we believe can have the most benefit to our system, as well as consider edge cases and how to deal with them if possible. The methods we considered are as follows:

- Improve detection via adversarial training - take images the network has failed detection on, and add them to the dataset. hopefully creating a more robust dataset.
- Improve detection using depth information - use depth as either a feature extractor, or to calculate height.
- piece-tracking using square-occupancy grids
- using the stereo cameras as additional cameras for the detection.

We then decided to examine and chart several edge cases, and how each method would deal with them.

Figure 25: Table of different methods to deal with the problem.

Edge Case/Method	Retraining	Depth – based piece identification	Use of Stereo cameras as additional inputs	Square Tracking
Wrong-ID	Should ideally not occur/be rare after retraining, no other recourse.	Depth based identification should correct it assuming that depth resolution is sufficient, and that the pieces have distinct heights (not always in our set).	Stereo Camera predictions when weighted should ideally correct the missed prediction.	Square tracking is completely robust to misidentifications so long as the color of the pieces is correctly identified
Depth Noise	Depth noise does not adversely affect this method.	Could severely hurt performance. Smoothing algorithms can help correct it.	Depending on noise origin could harm performance of stereo cameras since they're often used as depth sources. Otherwise impervious.	Depth noise does not adversely affect this method.
Label Flicker	Should ideally not occur/be rare after retraining, no other recourse.	Depth based identification should correct it assuming that depth resolution is sufficient, and that the pieces have distinct heights (not always in our set).	Stereo Camera predictions when weighted should ideally consistently help the model correctly classify.	Label Flicker does not robustly affect this method
Homography Issue (from movement)	Hurts localization, recalibration with SIFT and RANSAC required.	Hurts localization, recalibration with SIFT and RANSAC required.	Hurts localization, recalibration with SIFT and RANSAC required.	Hurts localization, recalibration with SIFT and RANSAC required.
Special Moves- promotion, en-passant, castling	Does not adversely affect the method	Does not adversely affect the method	Does not adversely affect the method	Method must have cases coded for the special moves – doable.
Promotion to different piece than queen	Does not affect method	Does not affect method	Does not affect Method	Must rely on identification to find the promoted piece. Rare occurrence though, could be trained for.

Based on the table and examining all of the methods, we decided to opt for both additional training and for the square tracking method. We knew the adversarial training had a possibility of not being fool-proof, but the addition of training samples could, at worst, improve our model and, at best, solve our detection issues entirely. Considering the relative ease of image labeling using Roboflow's platform, we decided to do so and added another 50 such images to our dataset.
Moreover, we decided to use the square-tracking method due to square-tracking's robustness to misidentification. In order to implement the square tracking algorithm we added a PieceTracker object that tracks the differences between square occupancy on the board every turn, and based on said differences determine which piece had moved, and which type of move did they perform, and then that information is relayed in the form of a move object to the GameState object's board.

6 Required Assumptions

Unfortunately, we cannot hope during the scope of the project to design a complete system that works in any state, so during the course of the project we had to choose requirements for our system to follow and limitations. Our system works so long as the following assumptions are fulfilled:

1. opencv's `findChessBoardCorners()` finds all the corners in the correct orientation, the illumination con-

ditions allow it to. Prior to setting all the pieces.

2. all the pieces remain in view of the camera, although partial occlusion is allowed.
3. all the pieces in the game have the correct starting chess configuration for the piece-tracker algorithm to work. Not all pieces must be in the game.
4. both players play legal moves sequentially barring the following: en-passant, castling, promotion. (since we did not account for them during the design of the system out of scope).
5. upon the completion of one move and prior to the start of another the button indicating a move has been completed must be pressed
6. the board and camera remain at a fixed angle relative to one another during the game.
7. the chess pieces and the board used should ideally be the ones our system was trained on, and we cannot guarantee an equivalent performance on other sets.

We'll note that a lot of the assumptions and errors we have stated can be overcome by various means. For instance, we can overcome the fixed-angle assumption by iteratively taking images of the current board, computing keypoints, computing a homography between the current board and the board prior to the move using RANSAC to filter outliers, and adjusting the homographic matrix by multiplying it with our previous one. The assumptions regarding castling or promotions can be dealt with by specifying the cases in code - for instance, for castling, one should see that the number of moved pieces is 2 of a single color and the conditions for castling are met. A lot of the assumptions regarding pieces can be overcome by training a bigger network on a bigger dataset and a more varied one for piece types to get robust detections. Move detection can be made by repeatedly checking whether the following two conditions are fulfilled: 1. that the board has transitioned from one legal state to another, 2. that no hands are touching any pieces. If they are, we can discern that a move has been made, and we can realize the system with a deep neural network to infer whether or not the second condition is fulfilled, and get the first by checking against GameState. For obstructions, we can use different cameras and homographies, and then associate detections from multiple cameras to the pieces. For errors relating to tracking, we can use a more robust tracking logic like DeepSort, implement Matching Cascade, tune the thresholds for the associator further, etc.
In summary, there are lots of methods to make a more robust system, but we must enforce these limitations due to scope constraints.

7 Final Results

Given the following constraints, our board performs admirably. Although we can see some flickering in the Kalman filter, the game state is consistently true; we are able to track the moves of the game state consistently and generate Stockfish-based suggestions.
We have fulfilled our requirements that we set out to do in the project definition under the assumptions. The figures below are taken from game recordings: we can see how in Figure 26 all the pieces are detected, and they are attributed to the correct squares. In the video it can be seen that the board is captured and represented as:

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

Where p signifies pawn, r rook, n knight, b bishop, k king, and q queen. Where capital letters denote a white piece, and the lower case letters represent black pieces.

Figure 26: game showcase: starting position

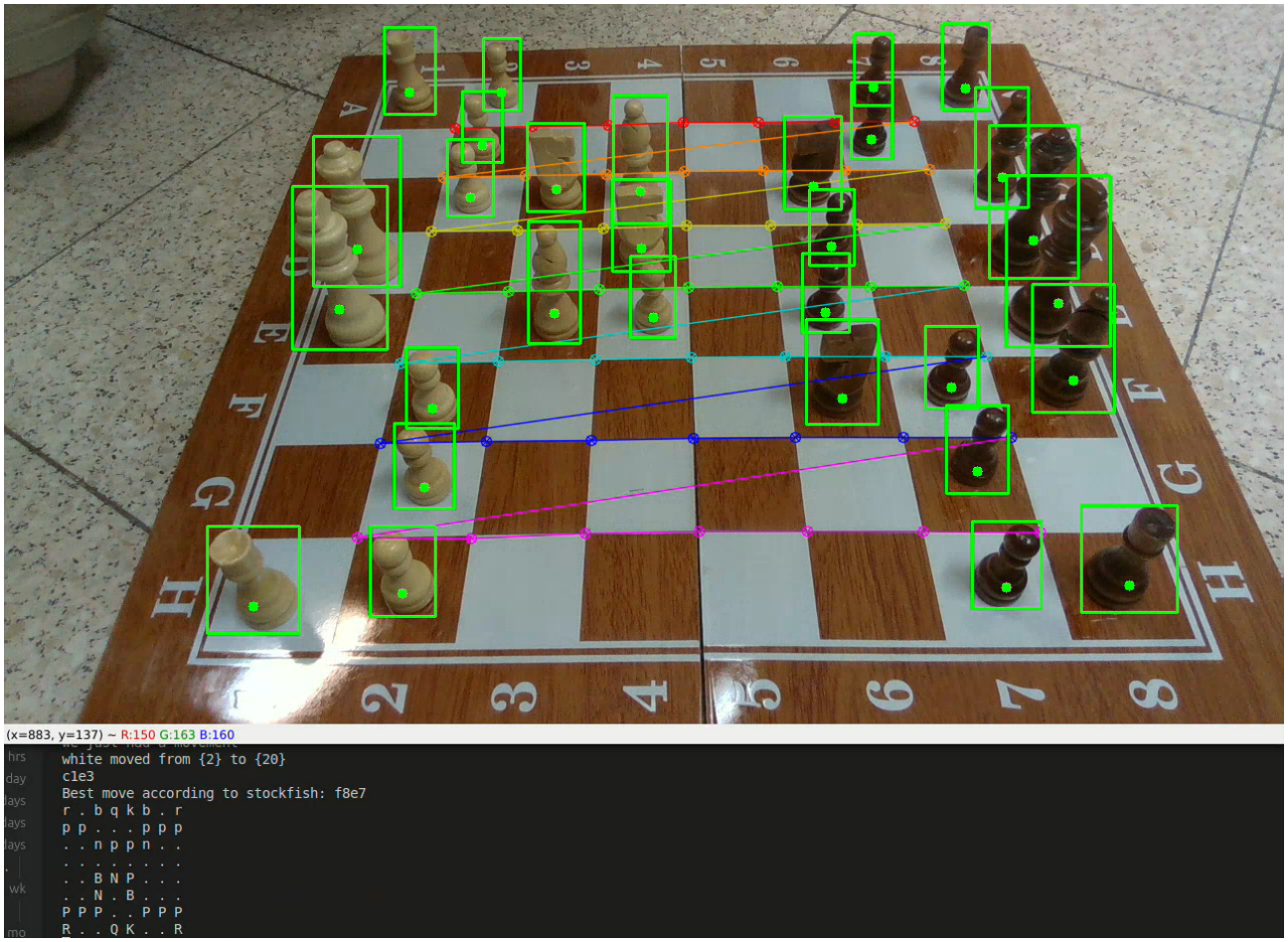


Figure 27: game Showcase: advanced position

and as we can see in the figure below (and the videos), we have the game in a very advanced position, after pieces were developed according to Stockfish suggestions and exchanged, that we can see that our system has maintained an accurate tracking even through captures, and we can generate Stockfish suggestions according to the board state.

8 Conclusion

The successful development of ChessCam validates the potential for computer vision systems to bridge physical and digital gaming experiences. While certain limitations remain, the system's core functionality demonstrates that automated chess analysis is not only feasible but can achieve high accuracy under appropriate conditions. The project's modular architecture and well-documented development process provide a solid foundation for future enhancements and serve as a valuable reference for similar computer vision applications in game analysis domains.

While our system achieved its core objectives, several limitations constrain its current applicability. The requirement for controlled lighting conditions and fixed camera positioning limits deployment flexibility. The system's inability to handle special chess moves such as castling, en passant, and promotion represents a significant functional gap that would need addressing for complete chess rule compliance.

The integration of multiple sophisticated technologies—from deep learning-based object detection to classical computer vision techniques and chess engine interfaces—showcases the power of interdisciplinary approaches in solving complex real-world problems. As computer vision technology continues to advance, systems like ChessCam will likely become increasingly robust and widely applicable, potentially revolutionizing how we interact with and analyze traditional board games.

This project allowed us the opportunity to experiment with techniques we previously only explored in the classroom, see the technical nuances of using them in practice. As well as learn about the fascinating world of multi-object tracking and object-oriented programming.

We enjoyed the project immensely and felt it was a gratifying, fascinating and fun challenge, and we're extremely happy and proud to have done it.

References

[1] "The guardian view on the chess boom: How rooks and knights captured the world," The Guardian. (Mar. 2025), [Online]. Available: https://www.theguardian.com/commentisfree/2025/mar/14/the-guardian-view-on-the-chess-boom-how-rooks-and-knights-captured-the-world.

[2] Ultralytics, *Intersection over union (iou)*, https://www.ultralytics.com/glossary/intersection-over-union-iou, 2024.

[3] S. Korman, *Course Slides: Algorithms and Applications in Computer Vision*, Lecture slides, Course 00460746, Technion – Israel Institute of Technology, Accessed via https://moodle24.technion.ac.il/course/view.php?id=2485, 2025.

[4] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[5] K. Solovey, *Lecture 7: State Estimation – Sensors, Bayes, and Kalman Filters*, Lecture slides, Course 00460213 – Mobile Robots, Technion – Israel Institute of Technology, Accessed via https://moodle24.technion.ac.il/course/view.php?id=2472, 2025.

[6] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in *2016 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2016, pp. 3464–3468. DOI: 10.1109/ICIP.2016.7533003. arXiv: 1602.00763.

[7] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *2017 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2017, pp. 3645–3649. DOI: 10.1109/ICIP.2017.8296962.

[8] Sachin, *Chess moves detection using computer vision algorithm*, https://youtu.be/ukNo9dZdUN4?si=fyam-i4SoNXl4YIy, 2023.

[9] M. Wolz, *Chessboard localization and piece recognition*, https://github.com/michaelwolz/ChessML, 2021.

[10] J. Gallagher and S. Nisan, *Chess board detection using yolov8 and roboflow*, https://blog.roboflow.com/chess-boards/, 2023.

[11] Spark, *I cheated in chess using computer vision*, https://www.youtube.com/watch?v=0dpA2QpjZDE, 2022.

[12] C. Belshe, *Chess piece detection capstone project*, https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1617&context=eesp, 2022.

[13] N. Aharon, R. Orfaig, and B.-Z. Bobrovsky, "Bot-sort: Robust associations multi-pedestrian tracking," *arXiv preprint arXiv:2206.14651v2*, 2022, Version 2, submitted 7 July 2022. DOI: 10.48550/arXiv.2206.14651.