<u>Project 4 Report</u>

Name: Priyanshu Ranka (002305396)

Semester: Spring 2025

Professor: Prof. Bruce Maxwell

Subject: Pattern Recognition and Computer Vision

1. Project Overview

This project focuses on **camera calibration, pose estimation, and augmented reality (AR)** to overlay a virtual 3D object onto a real-world scene. Using **OpenCV**, a **chessboard** was utilized as a reference for camera calibration, enabling the extraction of **intrinsic parameters** and **distortion coefficients**, which were stored in **CSV format** for efficient reuse.

For **pose estimation**, **solvePnP** computed the **rotation and translation vectors**, mapping 3D points to 2D image space. A **3D axis** was projected to validate pose estimation accuracy.

A **custom 3D virtual object**, modeled as a **spiral structure with square and diamond bases**, was projected onto the scene using **cv::projectPoints**. The object was dynamically drawn using **OpenCV's line-drawing functions**.

The real-time implementation captured **video frames, detected the chessboard, estimated pose**, and overlaid the virtual object. The system demonstrated **accurate projection alignment**, forming the foundation for **AR applications** in robotics, gaming, and industrial vision.

This project integrates **computer vision, 3D transformations, and AR visualization**, providing insights into real-world object tracking and projection.

2. Task Demonstrations

2.1 Detect and Extract Target Corners

<u>*Objective*</u>: The goal is to detect a calibration target (chessboard or ArUco markers) and extract its corner points. Using OpenCV functions like *findChessboardCorners, cornerSubPix, and drawChessboardCorners (or detectMarkers for ArUco),* the program should highlight the detected corners in a video feed. It should also display the number of detected corners and print the first corner's coordinates. Ensuring robustness in different lighting and orientations is crucial.

<u>*Implementation*</u>: The first step in the camera calibration process involves capturing multiple images of a chessboard pattern from different perspectives. In the provided implementation, a video feed is acquired using *OpenCV's VideoCapture (VideoCapture cap(0)),* and each frame is processed in real time. The function *findChessboardCorners()* is utilized to detect the chessboard pattern in the grayscale frames. Upon detection, the corners are stored in *cornerList*, and corresponding *3D world coordinates* are generated using *generateWorldFrame()*. The user can manually save calibration images by pressing *'s'*, which triggers the function *saveFrame(frame, ".", filename)*. A minimum of five such images must be captured before proceeding with the

calibration process. This step ensures that the dataset used for calibration adequately represents various viewing angles and positions, leading to more accurate intrinsic parameter estimation.

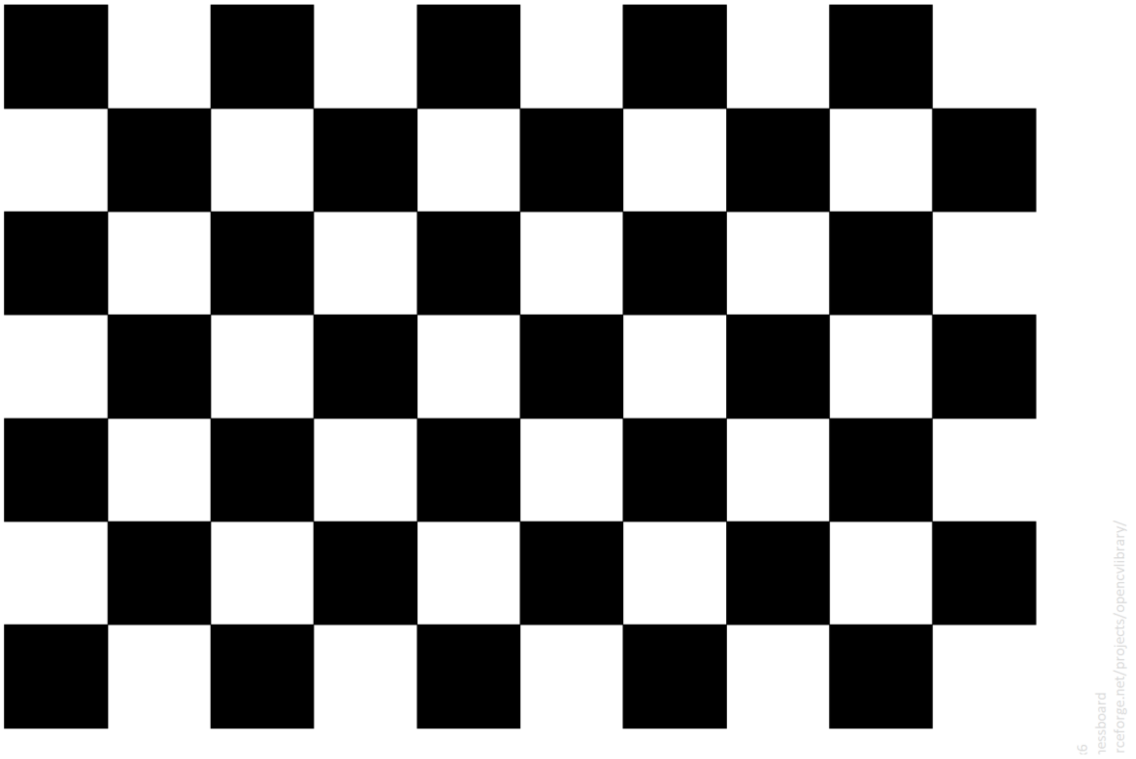


Figure 2.1.1: The chessboard used for calibration purposes.

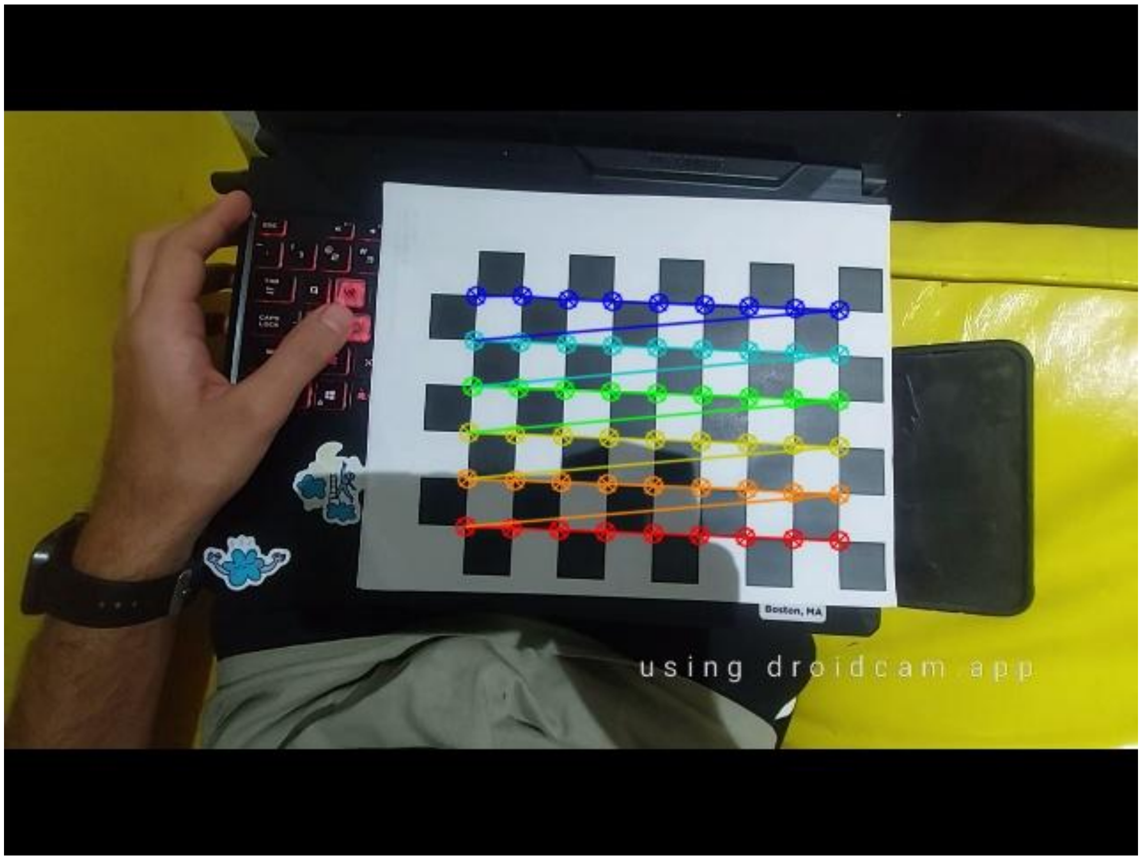


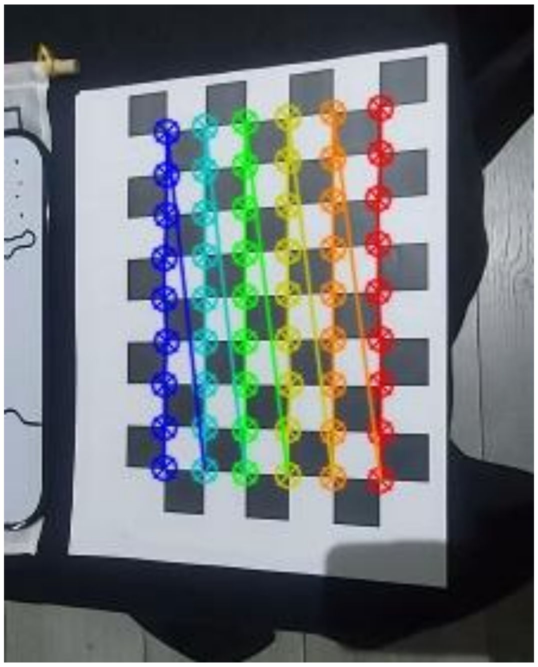
Figure 2.1.2: The detected Corners (9x6)

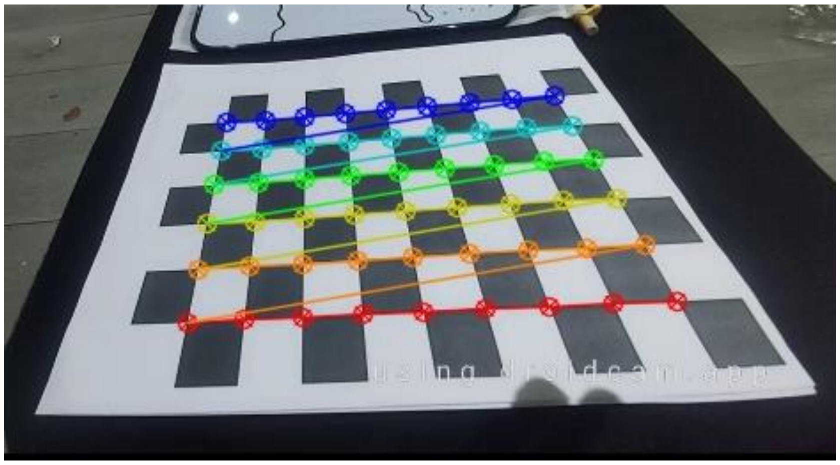
2.2 Select Calibration Images

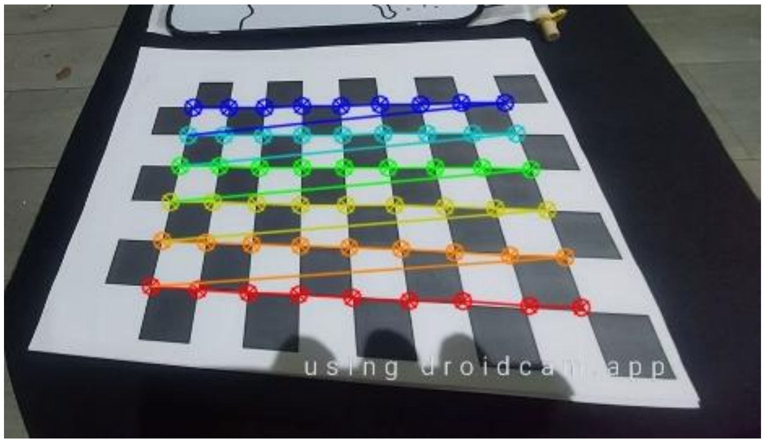
Objective: Users can manually select frames for calibration by pressing a key *('s')*. The program should store detected 2D image points and corresponding 3D world coordinates. The 3D points should be consistent regardless of target orientation, using a grid-based reference. These sets of points will be saved in vectors *(cornerList, pointSet, pointList)* for use in camera calibration. Ensuring correct row-column ordering is critical. The program may also store selected images for documentation. A sample calibration image with detected features should be included in the final report.

Implementation: Once enough images have been collected, the calibration process is initiated by pressing 'c'. The function calibrateCamera() is then employed to compute the *intrinsic camera matrix, distortion coefficients, and reprojection error* using the stored *2D image points* (*cornerList*) and corresponding *3D world points* (*pointList*). The calculated parameters are displayed on the console and saved in a CSV file via the function *saveCalibrationToCSV(cameraMatrix, distCoeffs, error)*. The computed camera matrix plays a crucial role in correcting lens distortions and improving accuracy in subsequent computer vision tasks. The reprojection error is also displayed to assess the quality of calibration, with lower values indicating better calibration accuracy.

Results:







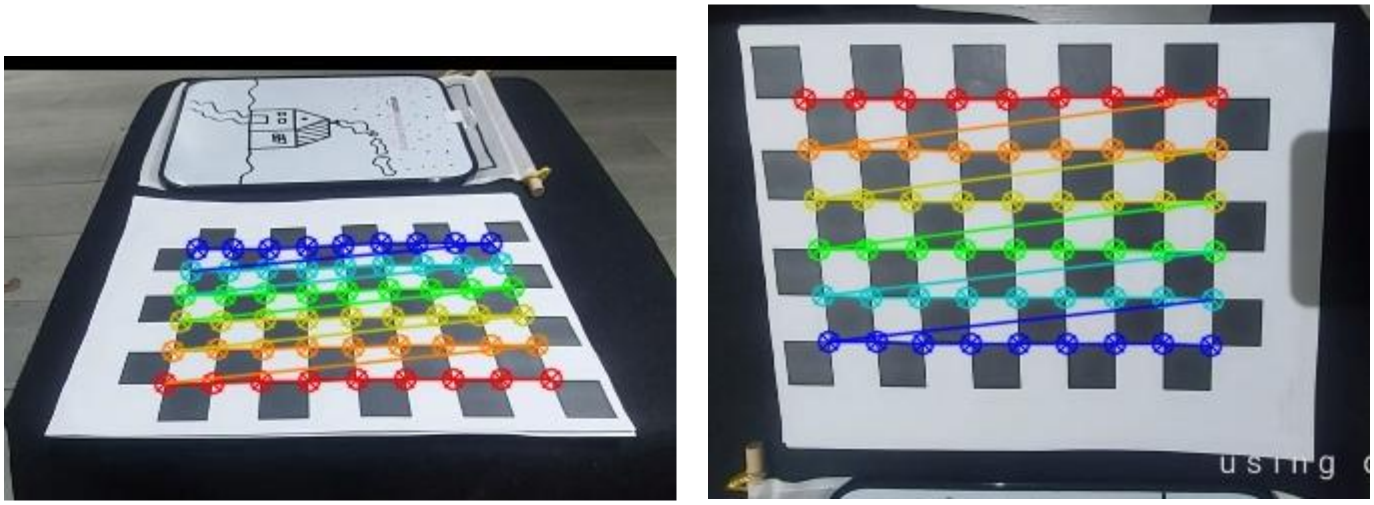


Figure 2.2.1: Example pictures used for calibration purposes.

2.3 Calibrate the Camera

Objective: Once at least five calibration frames are selected, the system should compute camera parameters using *calibrateCamera*. This includes estimating the camera matrix, distortion coefficients, rotation, and translation vectors. The camera matrix should be initialized with approximate values, assuming square pixels. The program should output the final calibration parameters and the reprojection error, which should ideally be below 1 pixel. Users should be able to save the calibration data to a file. The report should include the calibration matrix, distortion coefficients, and the final error estimation.

Implementation: Following successful calibration, the camera's *extrinsic parameters*, including its position and orientation relative to the chessboard, are determined using the *Perspective-n-Point (PnP) algorithm*. This is accomplished through the *solvePnP()* function, which takes the *3D object points (pointList[0]),* their corresponding *2D image points (cornerList[0]),* the computed *camera matrix,* and *distortion coefficients* as inputs. The function outputs the *rotation vector (rvec) and translation vector (tvec),* which describe the camera's pose relative to the reference chessboard. If the pose estimation is successful, the program displays a confirmation message and enables real-time pose visualization.

Results:

```
Camera Matrix
 246.527         0   317.573
       0   246.996   236.605
       0         0         1

Distortion Coefficients
                 -                      -
-0.04043   0.05699   0.006658   0.00325   0.05535

Reprojection Error
0.542968
```

Link to Video – LINK

2.4 Calculate Current Position of the Camera

Objective: This task involves determining the camera's real-time position relative to a detected target using *PnP (Perspective-n-Point).* The program should first load previously saved camera calibration parameters (camera matrix and distortion coefficients) from a file. It will then process a video stream to detect the calibration target (chessboard or ArUco markers) and extract corner positions. Using OpenCV's *solvePnP*, the program will compute the camera's *rotation and translation* relative to the target. The results should be displayed in real-time, showing how these values change as the camera moves. The report should analyze whether the output makes sense.

Implementation: To visualize the estimated camera pose dynamically, the program projects a *set of 3D coordinate axes* onto the captured image using the *projectPoints()* function. This function maps a local *(0,0,0) coordinate system* and its corresponding *X, Y, and Z axes* onto the image plane using the computed *rotation and translation vectors*. The projected axes are drawn as colored lines, where the X-axis is represented in red, the Y-axis in green, and the Z-axis in blue. These axes provide an intuitive visual representation of the camera's position relative to the chessboard. As the camera is moved in different directions, the orientation and positioning of these axes adjust accordingly, confirming the dynamic update of the estimated camera pose in real-time.

Results*: The output of the calibration makes perfect sense as the calculations are correct. Also, if changes must be made, that can be done easily using the 's' key to save more frames and calibrating on them according to our requirements. This is true for both moving object/frame and moving came.*

Link to Video- LINK

2.5 Project Outside Corners or 3D Axes

Objective: Using the *pose estimation* results from the previous step, the program should project at least four *3D chessboard corners* onto the image plane in real time using *cv::projectPoints()*. Alternatively, a *3D coordinate frame* (X, Y, Z axes) can be rendered on the target to aid in visualizing the camera's position and orientation. This ensures that the estimated pose is correctly mapped to the image. The program should dynamically update these projections as the camera or target moves, verifying the correctness of the reprojection by checking whether the points/axes align with the real-world chessboard.

Implementation: To visualize the accuracy of the pose estimation, a set of 3D axes was defined in world coordinates, with the origin at one corner of the chessboard. Using *OpenCV's projectPoints function*, these axes were transformed into 2D image coordinates. The *X-axis* was drawn in *red*, the *Y-axis* in *green*, and the *Z-axis* in *blue* using *cv::line*. These axes helped validate the correctness of the pose estimation by ensuring that they remained properly aligned as the camera moved around the scene. This step provided a *real-time AR reference* for future virtual object placement.

Results*: Link to Video -* LINK

2.6 Create a Virtual Object

Objective: A *3D virtual object* should be designed in world coordinates as a set of connected lines, initially with simple structures like a *pyramid or wireframe house*, before moving to more complex shapes. Each 3D point should be projected onto the image using *cv::projectPoints()* with the *rotation and translation matrices*, and lines should be drawn between the projected points in the image. The virtual object should remain *aligned with the* *chessboard's movement*, maintaining correct orientation as the camera moves, demonstrating an essential *augmented reality concept*.

Implementation: A *custom virtual object* was created as a 3D structure composed of connected points, forming a *spiral building-like shape* with stacked squares and diamonds. The object was defined in world space and then projected into the image using *cv::projectPoints*. The projected points were connected using *cv::line* to render the object in the correct perspective. The object dynamically adjusted as the camera moved, ensuring proper alignment due to the pose estimation. The design incorporated asymmetry to aid in debugging, making it easier to identify errors in projection and transformation.

Results: Link to Video- LINK

2.7 Detect Robust Features

Objective: A separate program should be implemented to detect and visualize *robust feature points* in a live video stream using techniques like *Harris corner detection, SURF, or SIFT*. The program should allow real-time visualization of detected features on a chosen pattern, experimenting with *different thresholds* to analyze feature robustness. This helps in understanding how these points can be used for augmented reality applications, such as object tracking or overlaying virtual objects on real-world scenes. Screenshots or videos should be captured to demonstrate the effectiveness of the feature detection.

Implementation: Feature detection was planned to be implemented separately using methods such as *Harris corners or SURF features*. This process would involve detecting key points in the image and highlighting them on a video stream. Different feature detection thresholds could be experimented with to determine robust feature sets. These feature points could then be used as an alternative approach to detecting surfaces and estimating pose, enabling more flexible augmented reality applications. This step could later be integrated with *pose estimation* to anchor virtual objects to detected features instead of relying solely on chessboard patterns.

Results: *Link to Video -* LINK

3. Reflections/Learnings

- **Importance of Accurate Camera Calibration** *(Task: Camera Calibration - CSV Data Loading)* Camera calibration is crucial for any computer vision application involving 3D reconstruction. Loading intrinsic parameters (camera matrix and distortion coefficients) from a **CSV file** instead of an XML file simplified data handling while ensuring precise transformations. Any misalignment in calibration directly affects pose estimation accuracy, leading to distorted projections.
- **Robust Pose Estimation with SolvePnP** *(Task: Pose Estimation - Chessboard Detection & 3D World Frame Mapping)* The **solvePnP** function effectively estimates the camera's pose by mapping **2D detected corners** to their corresponding **3D world coordinates**. The rotation and translation vectors extracted from this process allow accurate **camera-to-world transformations**, which are essential for AR applications. However, stability depends on good feature detection and an undistorted image.
- **Projecting 3D Points into 2D Image Space** *(Task: 3D Axis Projection & Virtual Object Placement)* The **cv::projectPoints** function was essential in converting 3D world coordinates into 2D image points. This function relies on **camera calibration and pose estimation** to ensure proper alignment. Without accurate projection, virtual objects may appear misplaced or incorrectly scaled, breaking the illusion of augmentation.
- **Challenges in Feature Detection and Chessboard Detection** *(Task: Feature Detection & Chessboard Corners in Pose Estimation)* Detecting chessboard corners can be **highly sensitive to lighting and occlusions**. The **findChessboardCorners** function occasionally failed in poor lighting, causing unstable pose estimation. A potential improvement would be integrating **Harris corners or ORB features** to detect key points more reliably, making the system more adaptable to different environments.
- **Understanding the Camera Coordinate System** *(Task: 3D Axis Projection & Virtual Object Construction)* The camera coordinate system follows the **right-hand rule**, where the X-axis extends **right**, the Y-axis extends **downward**, and the Z-axis moves **toward the camera**. This understanding was essential when defining the **3D axes** for visualization. Any misunderstanding of this coordinate system would lead to incorrectly projected objects.
- **Building Custom Virtual Objects Requires Geometric Planning** *(Task: Virtual Object - Spiral Structure Construction)* Designing a **virtual object** required careful arrangement of **3D points** to ensure correct projection. A **spiral-like structure** was chosen to create depth perception. Constructing the object required defining **base shapes (squares and diamonds)** and stacking them in a layered manner. Improperly defining these points could lead to misaligned or collapsed projections.
- **Real-Time Video Processing Efficiency Matters** *(Task: Real-Time Pose Estimation & Object Drawing in Video Feed)* Processing every frame in real-time required efficient computation. Direct **matrix operations (cv::Mat)** were used instead of loops to speed up transformations. Additionally, **downscaling the image** and reducing unnecessary computations (e.g., skipping redundant pose estimation) improved performance significantly while maintaining projection accuracy.
- **Depth Perception from 2D Projections Can Be Misleading** *(Task: 3D Axis Projection & Virtual Object Placement)* Although 3D objects were projected onto a 2D image, it was challenging to **perceive depth correctly** without proper shading or occlusion handling. Future improvements could involve using **OpenGL or shading techniques** to enhance depth perception, making the projected object appear more naturally integrated into the scene.
- **Frame Saving for Debugging and Offline Analysis** *(Task: Saving Video Frames for Debugging & Evaluation)* Implementing a **frame-saving feature** (saving images when pressing 's') allowed for debugging pose estimation results offline. This was crucial for analyzing errors and evaluating projection consistency across different frames without needing real-time reprocessing.
- **Practical Applications of Augmented Reality** *(Task: Overall Project - Combining Pose Estimation, Projection, and Virtual Object Rendering)* The project demonstrated a foundational **AR pipeline**, where real-world surfaces were detected, and virtual objects were projected accurately. This learning could be extended to applications such as **AR navigation, medical visualization, and industrial training**, showcasing the power of computer vision in merging digital and physical worlds.

4. Acknowledgements

I would like to express my gratitude to:

- **Professor Bruce Maxwell** for guidance and insightful lectures in **Pattern Recognition and Computer Vision (PRCV)**.
- **Teaching Assistants** for their support in understanding OpenCV and deep learning concepts.
- **Online Resources** (OpenCV documentation, research papers, and GitHub repositories) provided valuable insights into **image retrieval algorithms and deep network embeddings**.

This project has been an enriching experience, deepening my understanding of **image retrieval, feature extraction, and augmented-reality.**