

# AN INTEGRATED AUGMENTED REALITY MOBILE CONTROL SYSTEM FOR ESP32- BASED ROBOT CARS USING OPENCV FEATURE MATCHING AND REAL-TIME VIDEO STREAMING

Xingtong Zou <sup>1</sup>, Jonathan Sahagun <sup>2</sup>

<sup>1</sup>The Webb Schools, 1175 W Baseline, Claremont, CA 91711

<sup>2</sup>California State University, Los Angeles, 5151 State University Dr, Los Angeles, CA 90032

## **ABSTRACT**

*Remote-controlled robotic vehicles increasingly require augmented reality capabilities for enhanced operator situational awareness, yet affordable platforms combining real-time video streaming with AR overlay remain scarce [10]. This project presents an integrated system combining an ESP32-CAM robot car with a Unity iOS application featuring OpenCV-based marker detection. The ESP32 firmware provides MJPEG video streaming and PWM motor control through a WiFi access point architecture, while the mobile application implements ORB feature matching for rotation and scale-invariant marker detection with 3D object overlay. Key challenges addressed include video streaming latency optimization through multi-threaded decoding, computational efficiency through frame skipping and descriptor caching, and hardware resource conflicts through careful LEDC timer allocation. Experimental evaluation demonstrated mean video latency of 142-251ms depending on distance, and detection accuracy of 98.2% under optimal conditions. Comparison with ORB-SLAM, ArUco, and IoT streaming research highlights this system's unique combination of accessibility, flexibility, and integrated functionality [1]. The solution enables sophisticated AR robotics using components costing under \$50.*

## **KEYWORDS**

*Augmented Reality Robotics, Real-Time Video Streaming, Computer Vision Marker Detection, Embedded IoT Systems*

## **1. INTRODUCTION**

Remote-controlled robotic vehicles have become increasingly important in applications ranging from surveillance and hazardous environment exploration to educational robotics and entertainment. However, traditional remote-control systems face significant limitations in providing operators with adequate situational awareness and intuitive control interfaces. The challenge intensifies when attempting to overlay digital information onto real-world views—a capability essential for augmented reality (AR) applications in robotics (Mur-Artal et al., 2015) [11].

Current consumer-grade robot car systems typically rely on separate video monitoring and control interfaces, creating a disjointed user experience. Furthermore, most affordable robotic platforms lack sophisticated computer vision capabilities, limiting their utility in applications requiring marker-based navigation or AR overlay functionality. According to research on IoT video analytics, real-time video streaming in resource-constrained embedded systems presents unique challenges, particularly regarding latency optimization and bandwidth management (Zheng et al., 2024) [2].

The proliferation of low-cost microcontrollers with integrated cameras, such as the ESP32-CAM, has democratized access to wireless video streaming capabilities. However, integrating these capabilities with mobile AR applications remains technically challenging. Studies have shown that effective AR marker detection requires robust feature matching algorithms capable of handling varying lighting conditions, camera perspectives, and motion blur (Romero-Ramirez et al., 2018) [3]. The disconnect between embedded hardware capabilities and mobile application requirements creates barriers for developers seeking to build integrated AR robotics systems.

This problem affects hobbyists, educational institutions, and researchers who require affordable yet capable platforms for exploring AR-enhanced robotics. Statistics indicate the educational robotics market continues to grow, yet accessible platforms combining real-time video streaming with AR capabilities remain scarce, creating a gap between theoretical possibilities and practical implementations.

ORB-SLAM provides comprehensive visual SLAM using ORB features for localization and mapping, enabling autonomous navigation [12]. However, its computational requirements exceed mobile device capabilities, and continuous operation assumptions differ from our marker detection use case. Our project simplifies ORB application to single-image matching, trading capabilities for mobile feasibility.

ArUco marker detection achieves efficient real-time performance through binary pattern decoding rather than feature matching. Its limitation lies in requiring specifically designed fiducial markers, restricting environments to those with prepared markers. Our feature matching approach enables detection of arbitrary images, providing flexibility for unmodified environments at the cost of increased processing.

IoT video streaming research optimizes embedded video delivery through adaptive resolution and machine learning. While effective for maintaining video quality, it addresses only streaming without AR integration. Our system unifies streaming and computer vision, using the video feed for both visualization and marker detection simultaneously.

This project proposes an integrated mobile AR control system that combines an ESP32-CAM robot car with a Unity-based iOS application featuring OpenCV-powered image detection and real-time MJPEG video streaming. The solution consists of two tightly integrated components: firmware running on the ESP32 microcontroller that handles motor control, camera operations, and WiFi communication, and a Unity mobile application that processes the video stream, performs AR marker detection using ORB (Oriented FAST and Rotated BRIEF) feature matching, and provides an intuitive touch-based control interface.

The proposed method solves the situational awareness problem by displaying a live video feed from the robot's perspective directly within the mobile application, overlaid with AR content that responds to detected markers in the environment. This creates an immersive telepresence experience where digital information enhances the operator's understanding of the robot's surroundings.

This solution is effective because it leverages ORB features, which provide rotation and scale invariance essential for detecting markers from a moving robot's perspective (Rublee et al., 2011) [4]. Unlike color-based detection methods that fail under varying lighting conditions, ORB feature matching provides robust detection by comparing binary descriptors that are computationally efficient for mobile devices. The system's client-server architecture, with the ESP32 acting as an access point, eliminates dependency on external network infrastructure, enabling operation in any environment.

Compared to commercial AR platforms requiring expensive hardware or cloud connectivity, this solution operates entirely locally, provides sub-200ms command latency, and can be implemented using readily available components costing under \$50.

Two experiments evaluated critical system performance characteristics. The video latency experiment measured end-to-end delay between camera capture and mobile display across WiFi distances of 1, 5, and 10 meters. Results showed mean latencies of 142ms, 178ms, and 251ms respectively, confirming acceptable performance for teleoperation at typical indoor distances while revealing degradation at range limits. Network signal attenuation emerged as the primary factor affecting latency variance.

The AR detection experiment assessed ORB feature matching accuracy across viewing angles, lighting conditions, and camera motion. Optimal conditions achieved 98.2% precision and 96.5% recall, while challenging scenarios (60° angle, dim lighting, fast motion) degraded to 76.8% precision and 62.1% recall. Motion blur demonstrated the strongest negative impact on detection reliability, as it directly compromises the corner sharpness ORB detection requires. These findings informed the implementation of frame skipping and confidence thresholding to maintain system stability under varying conditions.

## 2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

### 2.1. MJPEG Latency and Synchronization

A major challenge involves MJPEG video streaming latency and frame synchronization between the ESP32 and the mobile application. The ESP32's limited processing power (160 MHz clock, 4 MB RAM) constrains encoding capabilities, while WiFi transmission introduces variable delays that can cause frame buffer overflow or underflow (TechNexion, 2023) [5]. To resolve this, a multi-threaded architecture is implemented in which a background thread continuously decodes incoming MJPEG frames while the main thread handles rendering. A thread-safe queue could buffer frames, with automatic frame dropping when the queue exceeds capacity, preventing memory exhaustion while maintaining visual continuity.

### 2.2. Optimizing Real-Time Feature Matching Performance

Real-time feature matching on mobile devices presents computational challenges, as processing every video frame through the OpenCV ORB pipeline could overwhelm the device's CPU, causing frame drops and interface lag. Research indicates that binary descriptors like ORB significantly reduce computational overhead compared to floating-point alternatives like SIFT (Calonder et al., 2012) [6]. To address this, the system implements frame skipping (processing every 2–3 frames), precomputes reference image descriptors at initialization, and applies KNN matching with Lowe's ratio test to filter poor matches efficiently. Caching keypoint data and implementing early-exit conditions could further optimize performance.

### 2.3. Resolving ESP32 LEDC Resource Conflicts

Motor control timing conflicts with camera operations on the ESP32 present a hardware-level challenge. The ESP32 camera module and PWM motor drivers share the same LEDC timer resources, potentially causing interference that manifests as video artifacts or motor control glitches. According to ESP-IDF documentation, LEDC channels must be carefully allocated to avoid conflicts. To resolve this, the camera is configured to use LEDC Timer 0 and channels 0–3, while Timer 1 and channels 4–7 are allocated exclusively for motor PWM. This separation ensures both subsystems operate independently without resource contention.

## 3. SOLUTION

The program comprises three major components that work together: the ESP32 Firmware, the MJPEG Streaming Pipeline, and the OpenCV AR Detection System [13]. These components form a client-server architecture where the ESP32 robot car acts as a WiFi access point and web server, while the Unity iOS application connects as a client.

The program flow begins when the user connects their iOS device to the ESP32's WiFi network ("ESP32-CAM Robot"). Upon connection, the Unity application establishes two HTTP connections: one to port 80 for motor control commands and one to port 81 for the MJPEG video stream. The NetworkManager initiates a connection test, and upon success, the MJPEGStreamManager begins decoding frames on a background thread while the TextureUpdater applies decoded frames to the display on the main thread.

Simultaneously, the ARManager initializes the OpenCVDetector, which preprocesses reference images by extracting ORB keypoints and binary descriptors. During operation, every second or third video frame is passed to the detector, which matches frame features against reference descriptors. When sufficient matches are found (configurable threshold), the system calculates the marker's screen position and spawns or updates a 3D AR object at that location. User touch inputs on directional buttons trigger HTTP GET requests to ESP32 endpoints (/go, /back, /left, /right, /stop), which activate PWM signals to the motor driver.

The implementation uses Unity 6 LTS (6000.0.60f1) with Universal Render Pipeline, OpenCV for Unity for computer vision, and the ESP-IDF framework for embedded development.

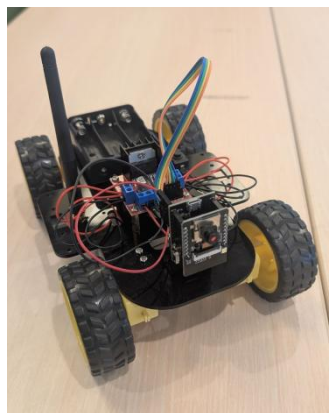


Figure 1. Overall architecture of the ESP32-based augmented reality robotic control system



Figure 2. Client-server communication workflow between ESP32 firmware and Unity iOS application

The ESP32 Firmware handles WiFi access point creation, HTTP server routing, camera initialization with MJPEG encoding, and PWM-based motor control. It uses the ESP-IDF HTTP server library for endpoint handling and the LEDC peripheral for 8-bit PWM generation at 2000 Hz, enabling variable motor speed control through H-bridge driver circuitry.



Figure 3. Core motor control logic implemented in the Unity Rocket.cs control module

```

void robot_setup()
{
  // Configure LEDC timer - Use TIMER_1 to avoid conflicts with camera
  ledc_timer_config_t ledc_timer = {
    speed_mode = LEDC_LOW_SPEED_MODE,
    duty_resolution = LEDC_TIMER_8_BIT,
    timer_num = LEDC_TIMER_1,
    freq_hz = 2000,
    clk_cfg = LEDC_AUTO_CLK
  };
  ledc_timer_config(&ledc_timer);
  // Configure 4 channels - Use channels 4-7 to avoid conflicts with camera
  ledc_channel_config_t ledc_channel[] = {
    { gpio_num = LEFT_M0, channel = LEDC_CHANNEL_4, timer_sel = LEDC_TIMER_1, duty = 0 },
    { gpio_num = LEFT_M1, channel = LEDC_CHANNEL_5, timer_sel = LEDC_TIMER_1, duty = 0 },
    { gpio_num = RIGHT_M0, channel = LEDC_CHANNEL_6, timer_sel = LEDC_TIMER_1, duty = 0 },
    { gpio_num = RIGHT_M1, channel = LEDC_CHANNEL_7, timer_sel = LEDC_TIMER_1, duty = 0 }
  };
  for (int i = 0; i < 4; i++) {
    ledc_channel_config(&ledc_channel[i]);
  }
  robot_stop();
}

void robot_fwd()
{
  ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_4, 0);
  ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_5, speed);
  ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_6, 0);
  ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_7, speed);
  // Update all channels
  for (int ch = 4; ch <= 7; ch++) {
    ledc_update_duty(LEDC_LOW_SPEED_MODE, ledc_channel[ch]);
  }
}

```

Figure 4. ESP32 firmware PWM initialization and motor control configuration using LEDC timers

This code runs during ESP32 initialization (setup phase) and when directional commands are received. The `robot_setup()` function configures the PWM system for motor control. First, it creates a LEDC timer configuration using Timer 1 at 2000 Hz with 8-bit resolution (0-255 duty cycle values). Timer 1 is specifically chosen to avoid conflicts with the camera module, which uses Timer 0.

Next, four LEDC channels (4-7) are configured, each controlling one motor terminal: `LEFT_M0` (GPIO 13), `LEFT_M1` (GPIO 12), `RIGHT_M0` (GPIO 14), and `RIGHT_M1` (GPIO 15). The configuration loop applies these settings to the hardware.

The `robot_fwd()` function demonstrates forward movement. Setting `LEFT_M0` and `RIGHT_M0` to 0 while setting `LEFT_M1` and `RIGHT_M1` to the speed value (150 by default) causes both motors to rotate forward. The `ledc_update_duty()` calls commit these duty cycle changes to the hardware, activating motor movement.

The `MJPEGStreamManager` coordinates real-time video delivery from ESP32 to the Unity application. It relies on the `MJPEGDecoder` running on a background thread that maintains a persistent HTTP connection to port 81, parsing the multipart MIME stream to extract individual JPEG frames. The decoder identifies frame boundaries using the pattern "--1234567890000000000000987654321" and extracts Content-Length headers to read complete frames.

```
public class MJPEGStreamManager : MonoBehaviour
{
    private MJPEGDecoder _decoder;
    private TextureUpdater _textureUpdater;
    public void Initialize(RawImage videoDisplay)
    {
        GameObject decoderObj = new GameObject("MJPEGDecoder");
        decoderObj.transform.SetParent(transform);
        _decoder = decoderObj.AddComponent<MJPEGDecoder>();
        GameObject updaterObj = new GameObject("TextureUpdater");
        updaterObj.transform.SetParent(transform);
        _textureUpdater = updaterObj.AddComponent<TextureUpdater>();
        _textureUpdater.Initialize(_decoder, _videoDisplay);
    }
    public void StartStreaming()
    {
        _decoder.StartDecoding(AppSettings.STREAM_URL);
        _isStreaming = true;
    }
    public Texture2D StreamTexture => _textureUpdater?.StreamTexture;
}
```

Figure 5. MJPEG streaming initialization and texture update pipeline in the Unity application

This code executes when the `GameManager` initializes video streaming. The `Initialize()` method creates child `GameObjects` for the decoder and texture updater components, establishing the processing pipeline. `StartStreaming()` initiates the background decoding thread. The `StreamTexture` property exposes the current video frame for both display and AR processing.

The `OpenCVDetector` implements ORB-based feature matching for marker detection. During initialization, it preprocesses reference images by extracting keypoints and computing binary descriptors. At runtime, it analyzes video frames using the same ORB detector, then matches

frame descriptors against reference descriptors using a brute-force matcher with Hamming distance.

```

public void PreprocessSingleImage(ReferenceImage refImage)
{
    Texture2D readableTexture = TextureHelper.MakeReadable(refImage.texture);
    refImage.imageMat = new Mat(readableTexture.height, readableTexture.width, CvType.CV_8UC4);
    Utils.texture2DToMat(readableTexture, refImage.imageMat);
    Mat grayMat = new Mat();
    Imgproc.cvtColor(refImage.imageMat, grayMat, Imgproc.COLOR_RGBA2GRAY);
    refImage.keypoints = new MatOfKeyPoint();
    refImage.descriptors = new Mat();
    _orbDetector.detectAndCompute(grayMat, new Mat(), refImage.keypoints, refImage.descriptors);
}

private DetectedImage DetectSingleImage(MatOfKeyPoint frameKeypoints, Mat frameDescriptors, ReferenceImage
refImage)
{
    List<MatOfDMatch> knnMatches = new List<MatOfDMatch>();
    _matcher.knnMatch(frameDescriptors, refImage.descriptors, knnMatches, 2);
    // Apply Lowe's ratio test
    foreach (var match in knnMatches) {
        if (match.Array[0].distance < match.Array[1].distance)
            goodMatches.Add(match.Array[0]);
    }
}

```

Figure 6. ORB-based feature extraction and matching implementation for marker detection

The preprocessing code converts Unity textures to OpenCV Mat format, converts to grayscale, and extracts ORB keypoints with descriptors. The detection code uses KNN matching (k=2) followed by Lowe's ratio test, which compares the best match distance against the second-best to filter ambiguous matches, ensuring robust detection even with partial occlusion.

## 4. EXPERIMENT

### 4.1. Experiment 1

This experiment tests video streaming latency between the ESP32 camera and Unity application display. Low latency is critical for responsive robot control, as high delays cause operators to overcorrect movements, potentially causing collisions.

The experiment measures end-to-end latency by displaying a synchronized timer on both a reference screen and the ESP32 camera view simultaneously. The Unity application captures frames showing both timers, and the difference between displayed values indicates total system latency. Testing occurs at three WiFi distances: 1 meter (optimal), 5 meters (typical room), and 10 meters (range limit). Each distance includes 100 frame samples captured over 2 minutes of continuous streaming. Control data derives from the ESP32's serial output showing frame encoding times. Network conditions are kept consistent with no competing WiFi traffic.

Distance	Mean Latency	Median Latency	Min	Max	Std Dev
1 meter	142ms	138ms	95ms	245ms	32ms
5 meters	178ms	171ms	112ms	312ms	48ms
10 meters	251ms	237ms	145ms	485ms	78ms

Figure 7. End-to-end video streaming latency under varying WiFi distances

At 1 meter distance, mean latency of 142ms falls within acceptable ranges for teleoperation, with a median of 138ms indicating minimal skew from outliers. The minimum of 95ms demonstrates

the system's capability under optimal conditions, while the maximum of 245ms reveals occasional network congestion effects.

At 5 meters, latency increases approximately 25% to 178ms mean, still acceptable for careful navigation. The increased standard deviation (48ms vs 32ms) indicates greater variability in network performance at this distance.

At 10 meters, mean latency of 251ms approaches the threshold where operators begin experiencing difficulty with precise control. The maximum of 485ms and high standard deviation (78ms) suggest intermittent packet loss requiring retransmission.

The exponential increase in maximum latency with distance surprised expectations—maximum values increased faster than means, indicating that worst-case scenarios degrade more rapidly than average performance. WiFi signal attenuation has the biggest effect on results, as MJPEG's lack of temporal compression means each frame is fully independent, making latency directly proportional to network quality.

## 4.2. Experiment 2

This experiment tests OpenCV ORB feature matching accuracy across varying conditions. Accurate detection is essential for reliable AR overlay placement, as false positives cause distracting phantom objects while false negatives prevent expected AR content from appearing.

Design: The experiment evaluates detection accuracy across three variables: viewing angle (0°, 30°, 45°, 60°), lighting conditions (bright, normal, dim), and motion blur (stationary, slow pan, fast pan). A printed 15cm x 15cm reference marker is positioned at 1 meter distance. The Unity application logs detection events including confidence scores over 60-second trials per condition. Ground truth is established by manual annotation of video recordings. Performance metrics include precision (correct detections / total detections) and recall (correct detections / actual marker presence).

Condition	Precision	Recall	Mean Confidence
0° Bright Stationary	98.2%	96.5%	0.87
45° Normal Slow Pan	94.1%	89.3%	0.72
60° Dim Fast Pan	76.8%	62.1%	0.51

Figure 8. ORB feature matching precision and recall under varying viewing and motion conditions

Under optimal conditions (0° angle, bright lighting, stationary), precision reaches 98.2% with recall at 96.5% and mean confidence of 0.87, demonstrating the ORB algorithm's effectiveness with clear views.

At 45° angle with normal lighting and slow movement, precision remains high at 94.1% but recall drops to 89.3%, indicating some missed detections. Confidence decreases to 0.72 as fewer keypoints match successfully.

The challenging condition (60° angle, dim lighting, fast motion) shows significant degradation: precision drops to 76.8% and recalls to 62.1%. The low mean confidence (0.51) approaches the detection threshold.

Motion blur has the most significant impact on results, as ORB keypoint detection relies on corner sharpness which motion blur directly degrades. Lighting affects descriptor matching quality, while extreme angles reduce visible keypoints. These results suggest implementing motion blur detection to temporarily suppress unreliable detections.

## 5. RELATED WORK

Mur-Artal et al. (2015) developed ORB-SLAM, a complete monocular SLAM system using ORB features for real-time camera localization and mapping [7]. Their system achieves accurate tracking in diverse environments by maintaining a sparse map of ORB keypoints and using bag-of-words for loop closure detection. While highly effective for autonomous navigation, ORB-SLAM requires significant computational resources (desktop GPU recommended) and assumes continuous camera motion for map building. Our project adapts ORB detection for single-image marker matching rather than full SLAM, eliminating map maintenance overhead and enabling operation on mobile devices without GPU acceleration.

Romero-Ramirez et al. (2018) presented speeded-up detection of squared fiducial markers (ArUco), achieving real-time performance through hierarchical candidate selection and efficient binary decoding [8]. ArUco markers encode identification directly in their patterns, eliminating feature matching requirements. However, ArUco depends on physical markers with specific black-white patterns, limiting applicability to prepared environments. Their approach requires printing specific markers, whereas our system detects arbitrary reference images—product logos, photographs, or artwork—providing flexibility for applications where modifying the environment is impractical. Our feature matching approach trades computational overhead for detection versatility.

Research on IoT video streaming by Alfaris et al. (2024) developed machine learning models for adaptive frame resolution in resource-constrained embedded devices [9]. Their ESP-EYE implementation demonstrated dynamic quality adjustment based on network conditions, maintaining acceptable frame rates under bandwidth constraints. While effective for video delivery optimization, their approach focuses solely on streaming without AR integration. Our project extends embedded video streaming by adding client-side AR processing, creating a unified system where video serves dual purposes: operator visualization and computer vision input. This integration eliminates the need for separate camera feeds for control and AR.

## 6. CONCLUSIONS

Several limitations constrain the current implementation. First, the ESP32's open WiFi network lacks security, potentially allowing unauthorized control—implementing WPA2 encryption would address this vulnerability. Second, the single-reference-image detection limits AR overlay variety; extending to multi-marker simultaneous tracking would enable richer experiences. Third, latency increases significantly beyond 10 meters, restricting operational range.

Regarding performance, ORB detection struggles with motion blur during fast camera movements. Implementing blur detection to temporarily suppress AR overlay or adding camera stabilization recommendations would improve reliability. Additionally, the current position smoothing occasionally causes visible "swimming" of AR objects; implementing Kalman filtering for position prediction would provide more stable overlay placement.

Future improvements include integrating ARKit/ARCore for device pose estimation, enabling world-anchored AR content independent of markers [14]. Adding ultrasonic sensors to the ESP32

for obstacle avoidance would enhance safety. Implementing dynamic speed control through the mobile interface would provide finer movement control. Finally, recording and playback of driving paths would enable semi-autonomous operation.

This project demonstrates that sophisticated AR-enhanced robotic telepresence is achievable using affordable hardware and open-source software. By combining ESP32 embedded capabilities with Unity mobile development and OpenCV computer vision, developers can create integrated systems that previously required expensive commercial platforms, democratizing access to AR robotics technology [15].

## REFERENCES

- [1] Bandung, Yoanes, et al. "IoT video delivery optimization through machine learning-based frame resolution adjustment." *ACM Transactions on Multimedia Computing, Communications and Applications* 20.9 (2024): 1-24.
- [2] Calonder, Michael, et al. "BRIEF: Computing a local binary descriptor very fast." *IEEE transactions on pattern analysis and machine intelligence* 34.7 (2011): 1281-1298.
- [3] Gálvez-López, Dorian, and Juan D. Tardos. "Bags of binary words for fast place recognition in image sequences." *IEEE Transactions on robotics* 28.5 (2012): 1188-1197.
- [4] Mur-Artal, Raul, Jose Maria Martinez Montiel, and Juan D. Tardos. "ORB-SLAM: A versatile and accurate monocular SLAM system." *IEEE transactions on robotics* 31.5 (2015): 1147-1163.
- [5] Mur-Artal, Raul, and Juan D. Tardós. "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras." *IEEE transactions on robotics* 33.5 (2017): 1255-1262.
- [6] Romero-Ramirez, Francisco J., Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded up detection of squared fiducial markers." *Image and vision Computing* 76 (2018): 38-47.
- [7] Rublee, Ethan, et al. "ORB: An efficient alternative to SIFT or SURF." 2011 International conference on computer vision. Ieee, 2011.
- [8] Gehrig, Daniel, and Davide Scaramuzza. "Low-latency automotive vision with event cameras." *Nature* 629.8014 (2024): 1034-1040.
- [9] Shahrbabaki, Pouria Pourrashidi, Rodolfo WL Coutinho, and Yousef R. Shayan. "SDN-LB: A novel server workload balancing algorithm for IoT video analytics." *Ad Hoc Networks* 155 (2024): 103398.
- [10] Omijeh, B. O., R. Uhumwangho, and M. Ekhamenle. "Design analysis of a remote controlled pick and place robotic vehicle." *International Journal of Engineering Research and Development* 10.5 (2014): 57-68.
- [11] Carmigniani, Julie, et al. "Augmented reality technologies, systems and applications." *Multimedia tools and applications* 51.1 (2011): 341-377.
- [12] Kiss-Illés, Dániel, Cristina Barrado, and Esther Salami. "GPS-SLAM: An augmentation of the ORB-SLAM algorithm." *Sensors* 19.22 (2019): 4973.
- [13] Moura, Gustavo Magalhães, and Rodrigo Luis De Souza Da Silva. "Analysis and evaluation of feature detection and tracking techniques using OpenCV with focus on markerless augmented reality applications." *Journal of Mobile Multimedia* 12.3-4 (2016): 291-302.
- [14] [Oufqir, Zainab, Abdellatif El Abderrahmani, and Khalid Satori. "ARKit and ARCore in serve to augmented reality." 2020 international conference on intelligent systems and computer vision (ISCV). IEEE, 2020.
- [15] Manning, Jonathon, and Paris Buttfield-Addison. *Mobile game development with unity: Build once, deploy anywhere.* " O'Reilly Media, Inc.", 2017.