2024-2025 Fall Semester Project Report

Department of Computer Engineering

CSE464 Digital Image Processing

Instructor: Dr. Habil KALKAN

Ziya Kadir TOKLUOGLU   Dogukan BAS

December 28, 2024

Abstract

This project presents a real-time human detection and counting system using core image processing techniques. A camera positioned overhead captures top-down frames of individuals passing through a designated gate. The proposed framework emphasizes low-level image operations—such as smoothing, thresholding, morphological transformations, and dynamic background subtraction—implemented directly in C++ to illuminate how these methods work at a fundamental level.

By carefully refining these operations, the system robustly distinguishes human silhouettes from noisy backgrounds, even under dynamic conditions. Further edge detection and region analysis techniques help accurately count individuals in each frame. Through this implementation, the project demonstrates how an understanding of underlying algorithms, combined with precise calibration and frame manipulation, can achieve real-time counting performance. The resulting pipeline offers a scalable, efficient, and cost-effective solution for access control, surveillance, or statistical data collection in various environments.

Contents

1 Introduction

The objective of this project is to develop a system that detects and counts humans passing through a designated gate using image processing techniques. The camera is positioned above the gate, capturing a top-down view to monitor movements and count individuals effectively. This project aims to deepen our understanding of image processing methods, aligning with the objectives of the lecture.

To achieve this, we focus on implementing core image processing operations manually, without relying on pre-built methods from additional libraries. By working directly with captured frames and utilizing only basic OpenCV functions for display, we aim to explore the underlying principles and workings of image processing techniques. Additionally, the project is implemented in C++ to ensure efficient processing speed, making it well-suited for this real-time application.

2 Image Processing Techniques

We explored many image processing techniques during this project, but not all were effective for our needs. While some methods were ultimately excluded, we believe it's important to mention them to reflect the breadth of our work.

- **Image Smoothing and Noise Reduction Techniques**
  - Median-Filter
  - Mean-Filter
  - Gaussian-Filter
- **Edge Detection**
  - Sobel Edge Detection
  - Canny Edge Detection
- **Morphological Operations**
  - Dilation
  - Erosion
- **Dynamic Background Substitutions**
  - MOG2
- **Other Operations**
  - Taking Threshold
  - Taking Grayscale
  - Subtraction Images

Lets Continue with methodology of this implementations.

3 Methodology

In this section, we outline the various image processing techniques employed in our study. Each technique is explained in detail, including mathematical formulations and filter representations where applicable. The methodologies are categorized into image smoothing and noise reduction, edge detection, morphological operations, dynamic background substitutions, and other essential image operations.

3.1 Image Smoothing and Noise Reduction Techniques

Image smoothing and noise reduction are fundamental preprocessing steps in image analysis. They help in enhancing image quality by reducing unwanted variations and preserving essential features.

3.1.1 Median Filter

The median filter is a non-linear filtering technique used to remove impulsive noise (salt-and-pepper noise) from an image. Unlike linear filters, the median filter preserves edges by replacing each pixel value with the median of the neighboring pixel values within a defined window.

Mathematical Formulation Given a grayscale image I and a window W of size $n \times n$ centered at pixel $I(i, j)$, the median filter computes:

$$I_{\text{median}}(i, j) = \text{median}\{I(k, l) \mid (k, l) \in W\}$$

Filter Representation Although the median filter is inherently non-linear and does not have a fixed matrix representation, it typically uses a square window such as 3×3:

$$W = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Here, e represents the central pixel whose value is replaced by the median of all elements in W.

3.1.2 Mean Filter

The mean filter is a linear smoothing technique that reduces random noise by averaging the pixel values within a specified window. While effective in noise reduction, it can blur edges due to the averaging process.

Mathematical Formulation For a grayscale image I and a window W of size $m \times m$, the mean filter computes:

$$I_{\text{mean}}(i, j) = \frac{1}{m^2} \sum_{k=-\frac{m-1}{2}}^{\frac{m-1}{2}} \sum_{l=-\frac{m-1}{2}}^{\frac{m-1}{2}} I(i + k, j + l)$$

Filter Representation A common 3×3 mean filter kernel is:

$$K_{\text{mean}} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3.1.3 Gaussian Filter

The Gaussian filter is a linear smoothing technique that applies a Gaussian function to weight the neighboring pixels, providing better preservation of edges compared to the mean filter.

Mathematical Formulation The Gaussian filter applies the following convolution:

$$I_{\text{Gaussian}}(i,j) = \sum_{k=-\frac{n}{2}}^{\frac{n}{2}} \sum_{l=-\frac{n}{2}}^{\frac{n}{2}} G(k,l) \cdot I(i+k, j+l)$$

where $G(k,l)$ is the Gaussian kernel defined as:

$$G(k,l) = \frac{1}{2\pi\sigma^2} e^{-\frac{k^2+l^2}{2\sigma^2}}$$

Filter Representation A typical 3×3 Gaussian kernel with $\sigma = 1$ is:

$$K_{\text{Gaussian}} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

3.2 Edge Detection

Edge detection is crucial for identifying significant transitions in intensity, which correspond to object boundaries within an image.

3.2.1 Sobel Edge Detection

The Sobel operator is a gradient-based method that emphasizes edges by computing the first derivative of the image intensity. It uses two 3×3 convolution kernels to detect horizontal and vertical changes. The Sobel operator not only detects the presence of edges but also provides the orientation of the edges, which is essential for further image analysis tasks.

Mathematical Formulation The Sobel operator computes the gradients in the x and y directions using the following convolution kernels:

$$G_x = K_{Sobel_x} * I$$

$$G_y = K_{Sobel_y} * I$$

where the convolution operator $*$ represents the application of the kernel to the image I.
The specific kernels for the Sobel operator are:

$$K_{Sobel_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad K_{Sobel_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

After computing the gradients, the gradient magnitude G and gradient direction θ are calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Detailed Edge Detection Process

1. Convolution with Sobel Kernels:

   The image I is convolved with both K_{Sobel_x} and K_{Sobel_y} to obtain the gradient images G_x and G_y. This process highlights the intensity changes in the horizontal and vertical directions, respectively.

2. Gradient Magnitude Calculation:

   The gradient magnitude G represents the strength of the edges. It is computed using the Euclidean distance formula:

   $$G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$$

   This combines the horizontal and vertical gradients to provide a comprehensive measure of edge strength at each pixel.

3. Gradient Direction Calculation:

   The gradient direction θ indicates the orientation of the edge. It is calculated using the arctangent function:

$$\theta(i,j) = \arctan\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$$

   This information is useful for subsequent steps like non-maximum suppression in more advanced edge detection algorithms.

4. Thresholding:

   To decide whether a pixel is part of an edge, a threshold is applied to the gradient magnitude G. Pixels with G above a certain threshold are considered edge pixels.

$$I_{\text{edge}}(i,j) = \begin{cases} 1 & \text{if } G(i,j) > T \\ 0 & \text{otherwise} \end{cases}$$

   Here, T is the threshold value chosen based on the desired sensitivity of edge detection.

5. Edge Localization:

   The combination of gradient magnitude and direction allows for precise localization of edges. By analyzing the direction θ, edges can be accurately traced and connected, reducing noise and false detections.

Filter Representation

$$K_{Sobel_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad K_{Sobel_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

3.2.2 Canny Edge Detection

The Canny edge detection algorithm is a robust multi-step process designed to detect edges with high accuracy while minimizing noise and false detections. It consists of four primary steps: noise reduction, gradient calculation, non-maximum suppression, and double thresholding with edge tracking.

Step 1: Noise Reduction To reduce noise and prepare the image for gradient calculation, a Gaussian filter is applied to the input image. This smooths out minor variations and ensures that the gradients are less sensitive to noise.

$$I_{\text{smoothed}}(i,j) = G_\sigma * I(i,j)$$

Here, G_σ is the Gaussian kernel defined as:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The size of the Gaussian kernel and the value of σ determine the degree of smoothing.

Step 2: Gradient Calculation After smoothing, the Sobel operator is used to compute the intensity gradients in the horizontal (G_x) and vertical (G_y) directions:

$$G_x = K_{Sobel_x} * I_{\text{smoothed}}, \quad G_y = K_{Sobel_y} * I_{\text{smoothed}}$$

where K_{Sobel_x} and K_{Sobel_y} are the Sobel kernels for horizontal and vertical gradients, respectively:

$$K_{Sobel_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad K_{Sobel_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The gradient magnitude G and direction θ are then calculated as:

$$G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$$

$$\theta(i,j) = \arctan\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$$

Step 3: Non-Maximum Suppression This step refines the edge detection by thinning edges to a single-pixel width. The algorithm examines the gradient direction θ at each pixel and suppresses any pixel value that is not a local maximum in the direction of the gradient.

For a pixel at (i,j), the gradient direction $\theta(i,j)$ is quantized into one of four primary directions: - 0° (horizontal), - 45° (diagonal up), - 90° (vertical), - 135° (diagonal down).

If the gradient magnitude $G(i,j)$ is not greater than the magnitudes of the two neighboring pixels along the gradient direction, it is set to zero:

$$G(i,j) = \begin{cases} G(i,j) & \text{if } G(i,j) > G_{\text{neighbors along } \theta} \\ 0 & \text{otherwise} \end{cases}$$

Step 4: Double Thresholding and Edge Tracking In this step, strong edges and weak edges are classified based on two thresholds, T_{high} and T_{low}: - Strong edges: $G(i,j) > T_{\text{high}}$. - Weak edges: $T_{\text{low}} < G(i,j) \leq T_{\text{high}}$. - Non-edges: $G(i,j) \leq T_{\text{low}}$.

The weak edges are retained only if they are connected to strong edges, a process called edge tracking by hysteresis. This ensures that only significant edges are preserved while eliminating isolated weak edges caused by noise.

$$E(i,j) = \begin{cases} 1 & \text{if } G(i,j) > T_{\text{high}} \text{ or connected to strong edges} \\ 0 & \text{otherwise} \end{cases}$$

3.3 Morphological Operations

Morphological operations process images based on their shapes, utilizing structuring elements to probe and transform the geometrical structure of objects within the image.

3.3.1 Dilation

Dilation adds pixels to the boundaries of objects, effectively expanding them. It is useful for closing small holes and connecting disjoint objects.

Mathematical Formulation Given a binary image A and a structuring element B, dilation is defined as:

$$A \oplus B = \{z \mid (B)_z \cap A \neq \emptyset\}$$

where $(B)_z$ is the translation of B by vector z.

Filter Representation A common structuring element is a 3×3 square:

$$B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3.3.2 Erosion

Erosion removes pixels from the boundaries of objects, effectively shrinking them. It is useful for eliminating small objects and separating connected objects.

Mathematical Formulation Given a binary image A and a structuring element B, erosion is defined as:

$$A \ominus B = \{z \mid B_z \subseteq A\}$$

Filter Representation Using the same 3×3 square structuring element:

$$B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3.4 Dynamic Background Substitutions

Dynamic background subtraction techniques are employed to differentiate moving objects from the background in video sequences, adapting to changes over time.

3.4.1 MOG2 (Mixture of Gaussians)

MOG2 models each background pixel as a mixture of several Gaussian distributions, allowing for the representation of complex backgrounds with varying illumination and movement.

Mathematical Formulation For each pixel, MOG2 maintains K Gaussian components, each characterized by mean μ_k, variance σ_k^2, and weight w_k. The probability of a pixel value x is given by:

$$P(x) = \sum_{k=1}^{K} w_k \cdot \mathcal{N}(x|\mu_k, \sigma_k^2)$$

where $\mathcal{N}(x|\mu_k, \sigma_k^2)$ is the Gaussian distribution.

Update Mechanism The parameters μ_k, σ_k^2, and w_k are updated based on whether the current pixel value matches any existing Gaussian component. If no match is found, a new Gaussian is introduced, and the least probable component is removed.

3.5 Other Operations

Additional image processing operations are utilized to enhance image features and prepare data for analysis.

3.5.1 Taking Threshold

Thresholding is a simple segmentation technique that converts a grayscale image into a binary image based on a threshold value T. Pixels with intensity above T are set to one class (e.g., foreground), and those below to another (e.g., background).

Mathematical Formulation

$$I_{\text{binary}}(i,j) = \begin{cases} 1 & \text{if } I(i,j) > T \\ 0 & \text{otherwise} \end{cases}$$

3.5.2 Taking Grayscale

Converting an image to grayscale reduces computational complexity by eliminating color information, retaining only intensity variations.

Mathematical Formulation For a color image with RGB channels, the grayscale conversion is typically performed using:

$$I_{\text{gray}}(i,j) = 0.2989 \cdot R(i,j) + 0.5870 \cdot G(i,j) + 0.1140 \cdot B(i,j)$$

3.5.3 Subtraction Images

Image subtraction involves computing the difference between two images, which is useful for detecting changes or motion between frames.

Mathematical Formulation Given two images A and B, the subtraction image S is:

$$S(i,j) = |A(i,j) - B(i,j)|$$

4 Experience

4.1 Challenges

4.1.1 Background Subtraction

In the implementation of background subtraction, the approach involved subtracting the previous frame from the current frame during each iteration. However, a challenge was encountered when individuals passed through the gate. During this process, the frame intermittently detected the ground surface, which became visible as individuals moved through. This issue arose due to the physical presence of substances, causing inaccuracies in detection.

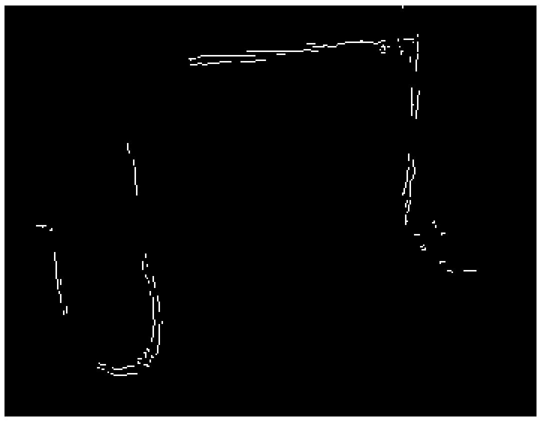

Figure 1:

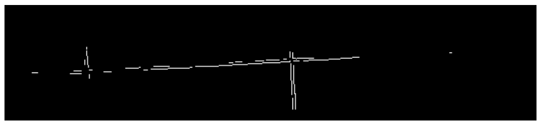

Figure 2:

According to the figures Figure1 and Figure2, the issue created an obstacle in the successful detection of humans. The detection algorithm was unable to differentiate between the ground surface and individuals passing through, leading to errors in human identification. This challenge highlighted the need for improved methods to account for such inconsistencies in background subtraction.

4.1.2 Background Subtraction 2 (MOG2)

The MOG2 algorithm, a dynamic background subtraction method, was implemented to address the challenges encountered in earlier approaches. However, several difficulties were identified during its application.

Firstly, the efficiency of the algorithm was observed to be quite low in the implemented code. This inefficiency significantly impacted the overall performance, making it unsuitable for real-time detection tasks.

Secondly, the algorithm failed to accurately capture human movement. Despite being a dynamic method, it struggled to differentiate between moving individuals and the background, resulting in frequent detection inaccuracies. These limitations underscore the need for further refinement and optimization to enhance the algorithm's practical applicability in such scenarios.

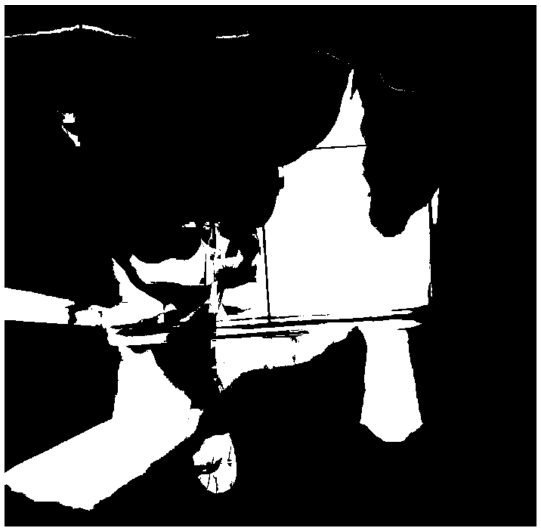

Figure 3:

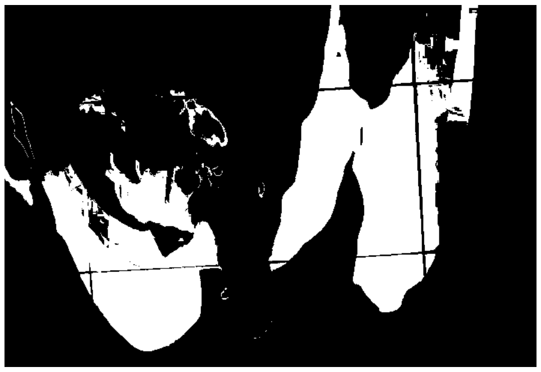

Figure 4:

According to the figures Figure3 and Figure4 several factors contributed to the challenges encountered during the implementation of the MOG2 algorithm. The variations in people's clothing, the ground patterns, and minor camera movements significantly affected the performance of the algorithm.

These factors not only increased the computational time required for processing but also resulted in a substantial decline in the accuracy of the detection. The algorithm struggled to adapt to these dynamic variations, leading to unreliable results and making it unsuitable for precise human detection in complex environments.

4.1.3 Background Subtraction 3

In this approach, an initial frame from the video was captured when no individuals were present. The Canny edge detection algorithm was then applied to this frame to extract the edges. The goal was to subtract the edges derived from the current frame during the video to retain only the edges corresponding to humans. While this method aimed to isolate human edges from the background, a significant issue was identified. The intersection of edges between humans and the ground resulted in the unintended removal of crucial human edges. Consequently, the extracted human edges contained gaps and holes, heavily influenced by the complexity and density of the ground surface edges. This limitation compromised the accuracy of edge detection and highlighted the challenges posed by intricate background patterns.

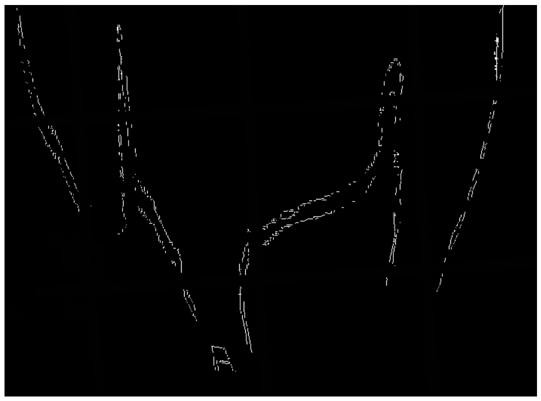

Figure 5:

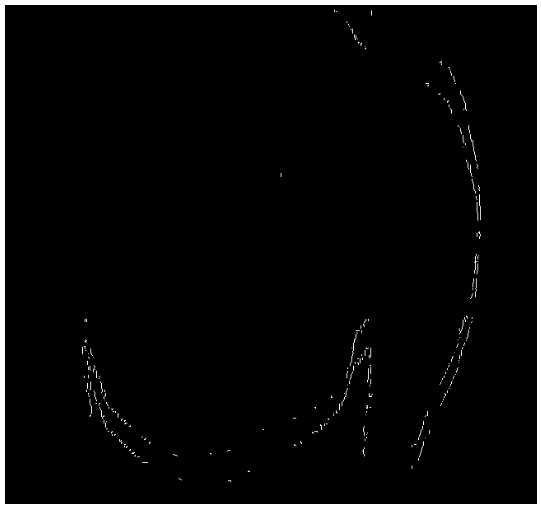

Figure 6:

As illustrated in Figure5 and Figure6, the limitations of this approach become evident. A close examination of the images reveals why the human edges could not be captured in a complete and accurate shape. The intersecting edges between humans and the ground, compounded by the ground's intricate patterns, result in gaps and irregularities in the detected human edges. These visual discrepancies highlight the difficulty in achieving precise edge extraction using this method.

4.1.4 Overall Background Subtraction

While each implementation presented its own set of challenges, the final approach demonstrated comparatively better results and is recommended for further exploration. Despite the inherent complexity of background subtraction and the absence of a universally flawless method, the results obtained through these experiments are valuable. Each implementation provided insights into the limitations of existing techniques, contributing to an iterative process of refinement. These findings highlight opportunities for improvement and offer a foundation for developing more robust methods in future studies.

4.2 The implementation steps of the Project

4.2.1 Taking Edges of Default Frame (Canny or Sobel)

The first step involved capturing a frame without any human presence and detecting its edges using either the Canny or Sobel edge detection algorithm. The primary purpose of this step was to create a mask that could be subtracted from the current frame to isolate the edges of humans present in subsequent frames.

To address the issue of minor camera instability, a dilation operation was applied to the detected edges. This step was crucial because small movements of the camera could cause ground edges to shift slightly, leading to inaccuracies. By performing dilation, the algorithm effectively reduced these errors, ensuring that ground edges were removed without introducing faults in the detection process.

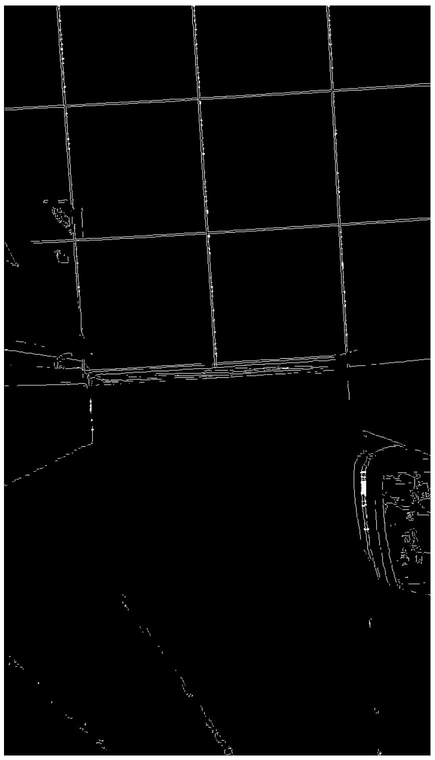
Figure 7:

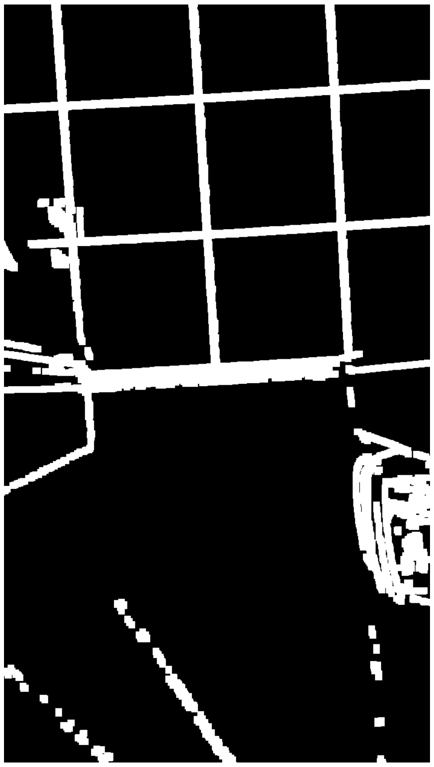
Figure 8:

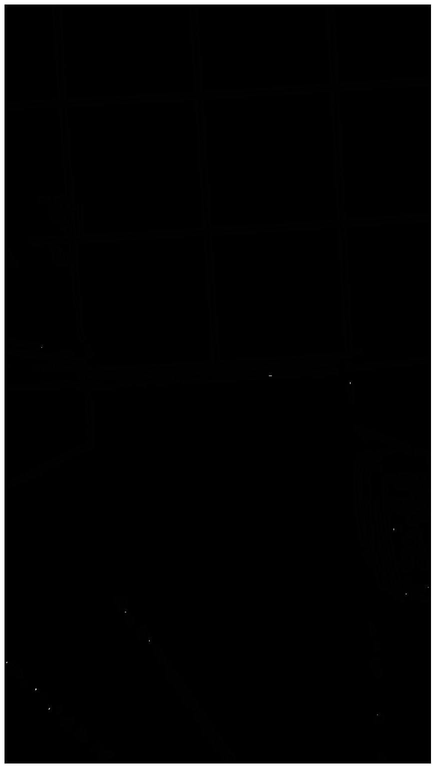
Figure 9:

As shown in Figure7, Figure8, and Figure9 the background subtraction process effectively eliminates the background edges. By using the edges of an initial frame (captured without any movement or dynamic elements) as a reference, the edges from subsequent frames were successfully subtracted.

This process resulted in a clean representation of the background, with minimal residual noise. The application of dilation helped address minor inconsistencies caused by slight camera movements, ensuring that edges from the static background were removed accurately. The minimal noise present in the resulting images demonstrates the effectiveness of this method in isolating and removing static background features.

4.2.2 Grayscale and Smoothing(Gaussion Blur)

The current frame was first converted to grayscale to simplify the processing by reducing the image to a single intensity channel. This step minimized computational complexity while preserving the essential features needed for analysis.

Subsequently, a Gaussian filter was applied to the grayscale image. This smoothing operation reduced noise and softened sharp edges, which could otherwise introduce inaccuracies during further processing steps. By removing random noise and sharpness, the Gaussian blur ensured a cleaner and more stable input for subsequent operations.

Figure 10:

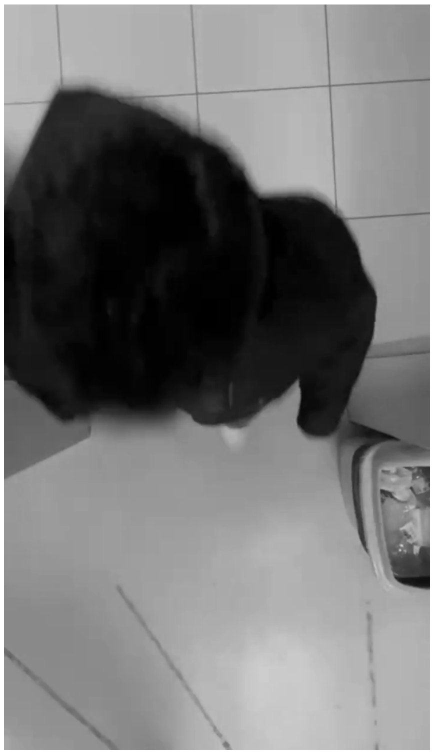

Figure 11:

4.2.3 Edge Detection and Substraction

In this step, edges were detected from the current frame using edge detection algorithms such as Canny or Sobel. The edges obtained from this frame were then subtracted from the previously created mask, which was derived from the initial frame without any dynamic elements.

This subtraction process effectively removed the static background edges from the current frame, isolating only the edges that represented changes or dynamic elements in the scene. By leveraging the pre-created mask, the approach minimized noise and ensured that only relevant edges were retained for further analysis.

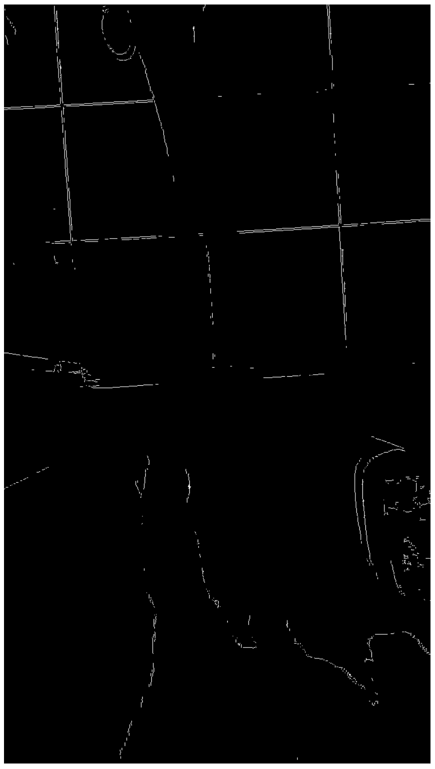

Figure 12:

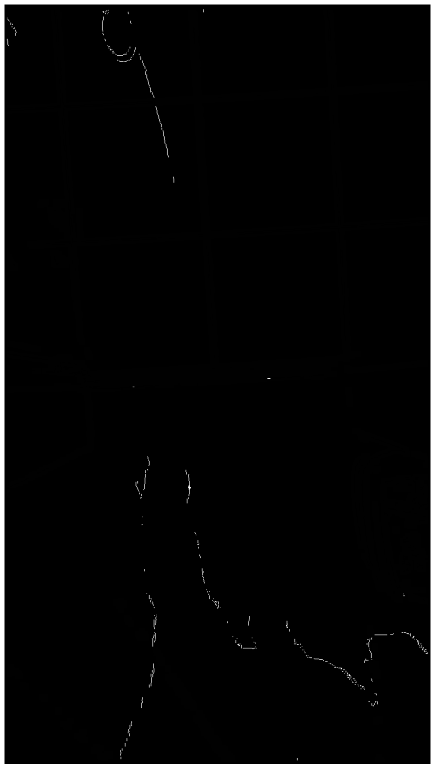

Figure 13:

As shown in Figure12 and Figure13, the edge detection and subtraction process effectively removes the static background edges, leaving the dynamic changes in the scene. However, despite the effectiveness of this method in isolating changes, there are significant limitations that impact the accurate detection of humans.

One of the main challenges is the incomplete separation of dynamic edges from the background due to overlapping or intersecting edges. The edges of humans and certain background features can intersect or blend, making it difficult to distinguish between them. Additionally, gaps or holes may appear in the detected edges, particularly in regions where the human edges are not sufficiently strong or continuous. These issues arise from variations in lighting, texture, and the inherent complexity of the scene, which affect the precision of edge detection and subtraction.

Furthermore, minor inconsistencies in camera stability and background patterns contribute to the inaccuracies, as the static background mask may not perfectly align with the current frame. These factors collectively limit the ability of this method to achieve flawless human detection.

4.3 Counting the Humans

To count the number of humans passing through the gate, two predefined regions (boxes) were determined in the frame. These regions were strategically positioned to monitor the movement across the gate area. For each frame, the edges within these boxes were detected and quantified.

The edge count in each region served as the basis for detecting the presence of a person. By analyzing the variation in edge counts over time, it was possible to identify when an individual passed through the gate. A threshold was established to distinguish between normal background fluctuations and significant changes caused by human movement.

This method provided a practical approach to human counting by leveraging edge detection and region-based analysis. However, the accuracy of this approach was influenced by factors such as edge completeness, lighting conditions, and the presence of overlapping objects, which occasionally led to undercounts or overcounts.

5 Conclusion

The project successfully implemented a real-time human detection and counting system using fundamental image processing techniques. By focusing on core operations such as edge detection, morphological transformations, and background subtraction, we demonstrated how low-level methods can be utilized to develop practical solutions for surveillance and monitoring applications. Despite the challenges encountered, the iterative nature of the development process provided valuable insights into the strengths and limitations of various approaches.

One of the most significant accomplishments was the manual implementation of edge detection and subtraction techniques, which allowed us to isolate dynamic elements in a frame. While effective to a certain extent, the project highlighted the difficulty of achieving consistent results under varying conditions, such as complex ground patterns, lighting fluctuations, and minor camera movements. These challenges underscored the importance of algorithmic refinement and the potential need for hybrid approaches that combine multiple techniques.

The experimentation with different background subtraction methods, including frame subtraction, MOG2, and edge-based subtraction, revealed their respective strengths and weaknesses. Frame subtraction provided a simple yet effective baseline, while MOG2 introduced dynamic adaptability at the cost of efficiency. The edge-based approach, though promising, faced difficulties in maintaining edge continuity, particularly in complex environments. These findings highlight the need for more robust solutions that can adapt to real-world scenarios with higher precision.

Additionally, the process of human counting using predefined regions and edge-based analysis demonstrated a practical application of the implemented techniques. While the approach was functional, its accuracy was influenced by factors such as edge overlap and gaps in detection. This points to the potential benefits of incorporating more sophisticated methods, such as machine learning-based object detection, in future iterations to improve reliability and scalability.

In conclusion, this project served as a comprehensive exploration of core image processing methods, providing hands-on experience with algorithm development and system implementation. The lessons learned from the challenges faced and solutions developed can serve as a foundation for further advancements. Future work could focus on addressing the identified limitations, integrating advanced techniques, and optimizing the system for real-time performance in diverse environments.

6 Appendix

6.1 Source Code

6.1.1 main.cpp

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>

#include "Filter.h"


using namespace std;
using namespace cv;

vector<int> getHistogramAndCompress(const cv::Mat& edgeFrame, int
    compressedSize = 100)
{
    // Ensure the input image is single-channel 8-bit
    CV_Assert(edgeFrame.type() == CV_8UC1);

    // 1) Compute the number of "white" pixels in each column of the
       original
    vector<int> originalHistogram(edgeFrame.cols, 0);
    for (int col = 0; col < edgeFrame.cols; col++)
    {
        int whiteCount = 0;
        for (int row = 0; row < edgeFrame.rows; row++)
        {
            // Count if pixel is 255 (edge/white)
            if (edgeFrame.at<uchar>(row, col) == 255)
            {
                whiteCount++;
            }
        }
        originalHistogram[col] = whiteCount;
    }

    // 2) Compress (downsample) the histogram to compressedSize columns
    vector<int> compressedHistogram(compressedSize, 0);

    // Simple binning approach
    for (int col = 0; col < edgeFrame.cols; col++)
    {
        // Map the column to one of the new compressed bins
        int binIndex = static_cast<int>(
            static_cast<double>(col) * compressedSize / edgeFrame.cols
        );
        // Accumulate the count
        compressedHistogram[binIndex] += originalHistogram[col];
    }

    // Return the final compressed histogram
    return compressedHistogram;
}

int main() {
    // Replace the camera capture with video file capture
    string videoFilePath = "Video3.mp4"; // Update this with your MP4
       file path
    VideoCapture cap(videoFilePath);

    if (!cap.isOpened()) {
        cerr << "Error: Could not open video file." << endl;
        return -1;
    }

    int counter = 0;


    Mat frame;
    cap >> frame;


    Mat first = Mat(frame.rows, frame.cols, CV_8UC1);

    CannyEdgeDetection(frame, frame.rows, frame.cols, 20, 80, first);

    cap >> frame;

    Mat second = Mat(frame.rows, frame.cols, CV_8UC1);

    CannyEdgeDetection(frame, frame.rows, frame.cols, 20, 80, second);


    // imshow("First Canny Edge Detection", first);
    // waitKey(0);
    // making the edges thicker
    for (int i = 0; i <5; i++) {
        Dilation(second, second.rows, second.cols);
    }

    // imshow("Second Canny Edge Detection", second);
    // waitKey(0);

    Mat edges = Mat(first.rows, first.cols, CV_8UC1);

    substract_grays(first, second, edges);


    // imshow("DifferencesMapping from the current", edges);
    // waitKey(0);

    Mat current_frame = Mat(frame.rows, frame.cols, CV_8UC3);
    Mat previous_frame = Mat(frame.rows, frame.cols, CV_8UC3);
    cout << "Frame size: " << frame.size() << endl;


    cap >> previous_frame;

    Mat differences = Mat(frame.rows, frame.cols, CV_8UC1);
    Mat CannyEdge = Mat(frame.rows, frame.cols , CV_8UC1);
    Mat grayImage;
    Mat differences2 = Mat(frame.rows, frame.cols, CV_8UC1);
    Mat differences3 = Mat(frame.rows, frame.cols, CV_8UC3);
    Mat differences4 = Mat(frame.rows, frame.cols, CV_8UC4);
    Mat differences5 = Mat(frame.rows, frame.cols, CV_8UC1);


    Mat current_frame2 = Mat(frame.rows, frame.cols, CV_8UC3);



    // These variable for the MOG2 algorithm
    float alpha = 0.01f; int K = 5; float T = 0.7f; float initialVar =
       900.0f;

    int average = 0;
    int average_2 = 0;


    bool is_human = false;
    bool is_human_2 = false;

    Mat gray = Mat(frame.rows, frame.cols, CV_8UC1);



    while (true) {
        // Capture a new frame from the video
        // for (int i = ; i < 2; i++) {
        cap >> current_frame;
        cap >> current_frame2;
        cap >> current_frame;

        gray = Grayscale(current_frame);


        GaussianBlurCustom(current_frame, current_frame.rows,
           current_frame.cols, 5, 1.4);

        take_differences(current_frame,previous_frame, differences3);
        differences4= Grayscale(differences3);
```

```cpp

        ApplyThreshold(differences4, differences5, 30, 255);
        Erosion(differences5, differences5.rows, differences5.cols);

        // ApplyThreshold3Channel(differences3, differences4, Vec3b(150, 150, 150), Vec3b(255, 255, 255));

        // CustomMOG2(god, god2, 0.01f, 5, 0.7f, 900.0f);
        // MOG2(gray, differences5, alpha, K, T, initialVar);

        CannyEdgeDetection(current_frame, current_frame.rows, current_frame.cols, 80, 140, CannyEdge);

        // Erosion(CannyEdge, CannyEdge.rows, CannyEdge.cols);

        // substract the mapping frame to current frame to lefting human edges alone
        substract_grays(CannyEdge, second, differences);

        // Camera cannot be placed certain position so we need to eliminate the sides
        eliminate_sides(differences, differences.rows, differences.cols, 10, differences2);

        for (int i = 0; i < 3; i++) {
            Dilation(differences2, differences2.rows, differences2.cols);
        }


        average = count_boxes(differences2, ((differences2.rows)/2 ) -5 - 400 , ((differences2.rows)/2 )+5 - 400);

        average_2 = count_boxes(differences2, ((differences2.rows)/2 )-5 + 400, ((differences2.rows)/2 )+5 + 400);


        cout << "average: " << average << endl;
        cout << "average_2: " << average_2 << endl;

        if (is_human == false && average > 300 ) {
            cout << "Human detected" << endl;
            is_human = true;
        } else if (is_human == true && average < 100) {
            cout << "Human left" << endl;
            is_human = false;
            counter++;
        }


        if (is_human_2 == false && average_2 > 300 ) {
            cout << "Human detected" << endl;
            is_human_2 = true;
        } else if (is_human_2 == true && average_2 < 100 ) {
            cout << "Human left" << endl;
            is_human_2 = false;
            counter--;
        }


        cout << "Counter: " << counter << endl;


        cout << "Counter: " << counter << endl;



        imshow("Final Frame", differences2);



        current_frame.copyTo(previous_frame);
        // counter += 1;
        // Exit the loop if the user presses the 'q' key
        if (waitKey(15) == 'q') {
            break;
        }
    }

    // Release the video file and close all OpenCV windows
    cap.release();
    destroyAllWindows();

    return 0;
}
```

Listing 1: main.cpp

6.1.2 Canny Edge Detection

```cpp
void Calculate_Gradient_magnitude(Mat& frame, int rows, int cols, int
   Sober_filter[3][3], Mat& gradientMagnitude, Mat& gradientDirection){

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            float Gx = 0, Gy = 0;

            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    Gx += frame.at<uint8_t>(i + k, j + l) * Sober_filter
                        [k + 1][l + 1];
                    Gy += frame.at<uint8_t>(i + k, j + l) * Sober_filter
                        [l + 1][k + 1];
                }
            }

            gradientMagnitude.at<float>(i, j) = sqrt(Gx * Gx + Gy * Gy);
            gradientDirection.at<float>(i, j) = atan2(Gy, Gx);
        }
    }

}



Mat& CannyEdgeDetection(Mat& frame, int rows, int cols, int lowThreshold
   , int highThreshold, Mat& output) {

    int Sober_filter[3][3] =
    {{-1, 0, 1},
     {-2, 0, 2},
     {-1, 0, 1}};

    Mat grayscale = Mat(rows, cols, CV_8UC1);
    Mat grayscale2 = Mat(rows, cols, CV_8UC1);

    Mat gradientMagnitude = Mat(rows, cols, CV_32FC1, Scalar(0));
    Mat gradientDirection = Mat(rows, cols, CV_32FC1, Scalar(0));
    Mat nonMaxSuppressed = Mat(rows, cols, CV_8UC1, Scalar(0));
    Mat edges = Mat(rows, cols, CV_8UC1, Scalar(0));


    // Median_Filter(frame, rows, cols);
    // Gaussion_Filter(frame, rows, cols);
    grayscale = Grayscale(frame); // take the grayscale image


    Calculate_Gradient_magnitude(grayscale, rows, cols, Sober_filter,
       gradientMagnitude, gradientDirection);


    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            float angle = gradientDirection.at<float>(i, j) * 180.0 /
               CV_PI;
            angle = angle < 0 ? angle + 180 : angle;

            float magnitude = gradientMagnitude.at<float>(i, j);
            float neighbor1 = 0, neighbor2 = 0;

            if ((angle > 0 && angle <= 22.5) || (angle > 157.5 && angle
               <= 180)) {
                neighbor1 = gradientMagnitude.at<float>(i, j - 1);
                neighbor2 = gradientMagnitude.at<float>(i, j + 1);
            } else if (angle > 22.5 && angle <= 67.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j + 1);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j - 1);
            } else if (angle > 67.5 && angle <= 112.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j);
            } else if (angle > 112.5 && angle <= 157.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j - 1);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j + 1);
            }

            if (magnitude >= neighbor1 && magnitude >= neighbor2) {
                nonMaxSuppressed.at<uint8_t>(i, j) = static_cast<uint8_t>(magnitude);
            } else {
                nonMaxSuppressed.at<uint8_t>(i, j) = 0;
            }
        }
    }

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            uint8_t value = nonMaxSuppressed.at<uint8_t>(i, j);

            if (value >= highThreshold) {
                edges.at<uint8_t>(i, j) = 255;
            } else if (value >= lowThreshold) {
                bool isConnectedToStrongEdge = false;
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {
                        if (nonMaxSuppressed.at<uint8_t>(i + di, j + dj) >= highThreshold) {
                            isConnectedToStrongEdge = true;
                            break;
                        }
                    }
                    if (isConnectedToStrongEdge) break;
                }
                edges.at<uint8_t>(i, j) = isConnectedToStrongEdge ? 255 : 0;
            } else {
                edges.at<uint8_t>(i, j) = 0;
            }
        }
    }

    output = edges.clone();

    return output;
}
```

Listing 2: Canny

6.1.3 Sober Edge Detection

```cpp
void Calculate_Gradient_magnitude(Mat& frame, int rows, int cols, int
   Sober_filter[3][3], Mat& gradientMagnitude, Mat& gradientDirection){

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            float Gx = 0, Gy = 0;

            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    Gx += frame.at<uint8_t>(i + k, j + l) * Sober_filter
                       [k + 1][l + 1];
                    Gy += frame.at<uint8_t>(i + k, j + l) * Sober_filter
                       [l + 1][k + 1];
                }
            }

            gradientMagnitude.at<float>(i, j) = sqrt(Gx * Gx + Gy * Gy);
            gradientDirection.at<float>(i, j) = atan2(Gy, Gx);
        }
    }

}



Mat& CannyEdgeDetection(Mat& frame, int rows, int cols, int lowThreshold
   , int highThreshold, Mat& output) {

    int Sober_filter[3][3] =
    {{-1, 0, 1},
     {-2, 0, 2},
     {-1, 0, 1}};

    Mat grayscale = Mat(rows, cols, CV_8UC1);
    Mat grayscale2 = Mat(rows, cols, CV_8UC1);

    Mat gradientMagnitude = Mat(rows, cols, CV_32FC1, Scalar(0));
    Mat gradientDirection = Mat(rows, cols, CV_32FC1, Scalar(0));
    Mat nonMaxSuppressed = Mat(rows, cols, CV_8UC1, Scalar(0));
    Mat edges = Mat(rows, cols, CV_8UC1, Scalar(0));


    // Median_Filter(frame, rows, cols);
    // Gaussion_Filter(frame, rows, cols);
    grayscale = Grayscale(frame); // take the grayscale image


    Calculate_Gradient_magnitude(grayscale, rows, cols, Sober_filter,
       gradientMagnitude, gradientDirection);


    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            float angle = gradientDirection.at<float>(i, j) * 180.0 /
               CV_PI;
            angle = angle < 0 ? angle + 180 : angle;

            float magnitude = gradientMagnitude.at<float>(i, j);
            float neighbor1 = 0, neighbor2 = 0;

            if ((angle > 0 && angle <= 22.5) || (angle > 157.5 && angle
               <= 180)) {
                neighbor1 = gradientMagnitude.at<float>(i, j - 1);
                neighbor2 = gradientMagnitude.at<float>(i, j + 1);
            } else if (angle > 22.5 && angle <= 67.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j + 1);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j - 1);
            } else if (angle > 67.5 && angle <= 112.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j);
            } else if (angle > 112.5 && angle <= 157.5) {
                neighbor1 = gradientMagnitude.at<float>(i - 1, j - 1);
                neighbor2 = gradientMagnitude.at<float>(i + 1, j + 1);
            }

            if (magnitude >= neighbor1 && magnitude >= neighbor2) {
                nonMaxSuppressed.at<uint8_t>(i, j) = static_cast<uint8_t
                   >(magnitude);
            } else {
                nonMaxSuppressed.at<uint8_t>(i, j) = 0;
            }
        }
    }

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            uint8_t value = nonMaxSuppressed.at<uint8_t>(i, j);

            if (value >= highThreshold) {
                edges.at<uint8_t>(i, j) = 255;
            } else if (value >= lowThreshold) {
                bool isConnectedToStrongEdge = false;
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {
                        if (nonMaxSuppressed.at<uint8_t>(i + di, j + dj)
                             >= highThreshold) {
                             isConnectedToStrongEdge = true;
                             break;
                        }
                    }
                    if (isConnectedToStrongEdge) break;
                }
                edges.at<uint8_t>(i, j) = isConnectedToStrongEdge ? 255 : 0;
            } else {
                edges.at<uint8_t>(i, j) = 0;
            }
        }
    }

    output = edges.clone();

    return output;
}
```

Listing 3: Sober

6.1.4 MOG2 Dynamaic background substraction

```

void MOG2(const Mat& frame,
                Mat& foregroundMask,
                float alpha = 0.01f,  // learning rate
                int K = 5,           // number of Gaussians per pixel
                float T = 0.7f,      // background threshold (ratio of summed weights)
                float initialVar = 900.0f)  // initial variance for newly replaced Gaussians
{
    static bool firstCall = true;
    static int rows = 0;
    static int cols = 0;

    static vector<Mat> means;
    static vector<Mat> variances;
    static vector<Mat> weights;

    if (firstCall || rows != frame.rows || cols != frame.cols) {
        firstCall = false;
        rows = frame.rows;
        cols = frame.cols;

        means.resize(K);
        variances.resize(K);
        weights.resize(K);

        for (int i = 0; i < K; i++) {
            means[i] = Mat(rows, cols, CV_32FC1, Scalar(0));
            variances[i] = Mat(rows, cols, CV_32FC1, Scalar(initialVar));
            weights[i] = Mat(rows, cols, CV_32FC1, (i == 0) ? Scalar(1.0f) : Scalar(0.0f));
        }
    }

    foregroundMask.create(rows, cols, CV_8UC1);

    for (int y = 0; y < rows; y++) {
        const uchar* rowPtr = frame.ptr<uchar>(y);
        for (int x = 0; x < cols; x++) {
            float pixelVal = static_cast<float>(rowPtr[x]);

            bool matched = false;
            int matchedIndex = -1;

            for (int i = 0; i < K; i++) {
                float m = means[i].at<float>(y, x);
                float v = variances[i].at<float>(y, x);
                float sigma = sqrt(v);

                if (fabs(pixelVal - m) <= 2.5f * sigma) {
                    matched = true;
                    matchedIndex = i;
                    break;
                }
            }

            if (matched) {
                float& w = weights[matchedIndex].at<float>(y, x);
                float& m = means[matchedIndex].at<float>(y, x);
                float& v = variances[matchedIndex].at<float>(y, x);

                w = (1.0f - alpha) * w + alpha;
                float diff = (pixelVal - m);
                m = m + alpha * diff;
                float diff2 = (pixelVal - m);
                v = v + alpha * (diff2 * diff2 - v);
            } else {
                int lowestIndex = 0;
                float lowestWeight = weights[0].at<float>(y, x);

                for (int i = 1; i < K; i++) {
                    float w_i = weights[i].at<float>(y, x);
                    if (w_i < lowestWeight) {
                        lowestWeight = w_i;
                        lowestIndex = i;
                    }
                }

                weights[lowestIndex].at<float>(y, x) = alpha;
                means[lowestIndex].at<float>(y, x) = pixelVal;
                variances[lowestIndex].at<float>(y, x) = initialVar;
            }

            float sumWeight = 0.0f;
            for (int i = 0; i < K; i++) {
                float& w = weights[i].at<float>(y, x);
                w = (1.0f - alpha) * w;
                sumWeight += w;
            }
            if (matched && matchedIndex >= 0) {
                weights[matchedIndex].at<float>(y, x) += alpha;
                sumWeight += alpha;
            }
            for (int i = 0; i < K; i++) {
                weights[i].at<float>(y, x) /= sumWeight;
            }
        }
    }

    for (int y = 0; y < rows; y++) {
        const uchar* rowPtr = frame.ptr<uchar>(y);
        uchar* outPtr = foregroundMask.ptr<uchar>(y);

        for (int x = 0; x < cols; x++) {
            vector<pair<float, int>> wVec;
            for (int i = 0; i < K; i++) {
                float w = weights[i].at<float>(y, x);
                wVec.push_back({w, i});
            }
            sort(wVec.begin(), wVec.end(), [](const pair<float, int>& a, const pair<float, int>& b) {
                return a.first > b.first;
            });

            float accumWeight = 0.0f;
            int backgroundCount = 0;
            for (int i = 0; i < K; i++) {
                accumWeight += wVec[i].first;
                backgroundCount++;
                if (accumWeight > T) {
                    break;
                }
            }

            float pixelVal = static_cast<float>(rowPtr[x]);
            bool isBackground = false;

            for (int i = 0; i < backgroundCount; i++) {
                int gIndex = wVec[i].second;
                float m = means[gIndex].at<float>(y, x);
                float v = variances[gIndex].at<float>(y, x);
                float sigma = sqrt(v);

                if (fabs(pixelVal - m) <= 2.5f * sigma) {
                    isBackground = true;
                    break;
                }
            }

            outPtr[x] = (isBackground ? 0 : 255);
        }
    }
}
```

Listing 4: MOG2

6.1.5 GaussionFilter

```

Mat& GaussianBlurCustom(Mat& frame, int rows, int cols, int kernelSize,
   double sigma) {
    // Create Gaussian kernel
    int halfSize = kernelSize / 2;
    vector<vector<double>> kernel(kernelSize, vector<double>(kernelSize)
       );
    double sum = 0.0;
    double s = 2.0 * sigma * sigma;

    for (int x = -halfSize; x <= halfSize; x++) {
        for (int y = -halfSize; y <= halfSize; y++) {
            double r = sqrt(x * x + y * y);
            kernel[x + halfSize][y + halfSize] = (exp(-(r * r) / s)) / (
               CV_PI * s);
            sum += kernel[x + halfSize][y + halfSize];
        }
    }

    // Normalize the kernel
    for (int i = 0; i < kernelSize; i++) {
        for (int j = 0; j < kernelSize; j++) {
            kernel[i][j] /= sum;
        }
    }

    // Apply Gaussian blur
    Mat* result = new Mat(rows, cols, frame.type());
    for (int i = halfSize; i < rows - halfSize; i++) {
        for (int j = halfSize; j < cols - halfSize; j++) {
            double sum = 0.0;
            for (int k = -halfSize; k <= halfSize; k++) {
                for (int l = -halfSize; l <= halfSize; l++) {
                    sum += frame.at<uint8_t>(i + k, j + l) * kernel[k +
                       halfSize][l + halfSize];
                }
            }
            result->at<uint8_t>(i, j) = static_cast<uint8_t>(sum);
        }
    }

    return *result;
}
```

Listing 5: Gaussion

6.1.6 Helper Functions

```
int count_boxes(Mat& frame, int rows1, int row2) {
    int count = 0;

    for (int i = rows1; i < row2; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (frame.at<uint8_t>(i, j) == 255) { // Calculate the average of all white indexes in every column
                count++;
            }
        }
    }

    return count;

}

Mat& Grayscale(Mat &frame) {
    // Create a new matrix for the grayscale image (single channel)
    // Mat grayImage(frame.rows, frame.cols, CV_8UC1); // Single channel (grayscale)
    Mat* grayImage = new Mat(frame.rows, frame.cols, CV_8UC1);

    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            Vec3b pixel = frame.at<Vec3b>(i, j);  // Access each pixel
            uint8_t grayValue = static_cast<uint8_t>(0.2989 * pixel[2] + 0.5870 * pixel[1] + 0.1140 * pixel[0]);
            (*grayImage).at<uint8_t>(i, j) = grayValue; // Store in the single channel grayscale image
        }
    }

    return *grayImage;
}
Mat& Median_Filter(Mat& frame, int rows, int cols){

    Mat* result = new Mat(rows, cols, CV_8UC3);

    for (int i = 1; i < rows-1; i++) {
        for (int j = 1; j < cols-1; j++) {

            vector<int> redValues;
            vector<int> greenValues;
            vector<int> blueValues;

            for(int k = -1; k <= 1; k++){
                for(int l = -1; l <= 1; l++){
                    redValues.push_back(frame.at<Vec3b>(i+k, j+l)[0]);
                    greenValues.push_back(frame.at<Vec3b>(i+k, j+l)[1]);
                    blueValues.push_back(frame.at<Vec3b>(i+k, j+l)[2]);
                }
            }

            sort(redValues.begin(), redValues.end());
            sort(greenValues.begin(), greenValues.end());
            sort(blueValues.begin(), blueValues.end());

            (*result).at<Vec3b>(i, j)[0] = redValues[4];
            (*result).at<Vec3b>(i, j)[1] = greenValues[4];
            (*result).at<Vec3b>(i, j)[2] = blueValues[4];
        }
    }

    return *result;
}

Mat& Mean_Filter(Mat& frame, int rows, int cols){

    Mat* result = new Mat(rows, cols, CV_8UC3);

    for (int i = 1; i < rows-1; i++) {
        for (int j = 1; j < cols-1; j++) {

            int sumR, sumG, sumB;

            for(int k = -1; k <= 1; k++){
                for(int l = -1; l <= 1; l++){
                    sumR += frame.at<Vec3b>(i+k, j+l)[0];
                    sumG += frame.at<Vec3b>(i+k, j+l)[1];
                    sumB += frame.at<Vec3b>(i+k, j+l)[2];
                }
            }

            (*result).at<Vec3b>(i, j)[0] = sumR / 9;
            (*result).at<Vec3b>(i, j)[1] = sumG / 9;
            (*result).at<Vec3b>(i, j)[2] = sumB / 9;
        }
    }

    return *result;
}
```

Listing 6: Gaussion

7 References

- Rong, W., Li, Z., Zhang, W., Sun, L. (2014, August). An improved CANNY edge detection algorithm. In *2014 IEEE International Conference on Mechatronics and Automation* (pp. 577-582). IEEE.

- Gao, W., Zhang, X., Yang, L., Liu, H. (2010, July). An improved Sobel edge detection. In *2010 3rd International Conference on Computer Science and Information Technology* (Vol. 5, pp. 67-71). IEEE.

- Trnovszký, T., Sýkora, P., Hudec, R. (2017). Comparison of background subtraction methods on near infra-red spectrum video sequences. *Procedia Engineering, 192*, 887-892.

- Young, I. T., Van Vliet, L. J. (1995). Recursive implementation of the Gaussian filter. *Signal Processing, 44*(2), 139-151.

- Justusson, B. I. (2006). Median filtering: Statistical properties. *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters*, 161-196.