ECE420 Embedded Digital Signal Processing

Final Report

Android-Embedded Security System

Paul Jablonski, Jun Hayakawa

{pjj3, jundh2} @ illinois.edu

December 16th, 2024

Table of Contents

1. Introduction

1.1 Problem Statement

Contemporary security systems often involve complex installation processes and multi-component setups to operate correctly. They frequently impose the challenges of data storage and management on small businesses and tend to be inefficient in their core objectives of securing areas and preventing crime, as they lack features to identify individuals passing by. Moreover, instead of storing only pertinent data, these systems can also waste significant amounts of data through wireless transmissions. Wireless security cameras are found to waste up to 400 GB of data per month, which for a smaller business can be financially significant [1].

1.2 Implemented Solution

The implemented solution to this issue is creating an easily-deployable and data-conservative Android system that can eliminate primary sources of storage waste. More specifically, this solution masks human subjects from the background of security footage through utilizing a calibration image with nobody in it. This calibration image is compared to live frames to determine differences in the foreground and the background - ultimately isolating the foreground. Rectangular object proposals are then bound for each isolated person in frame by creating a bounding box around edge clusters. This reduces the resolution of images being stored, as only the identified person and their cropped bounding box needs to be stored in the security system's database. The key idea is avoiding the storage of entire frames, and instead just storing key information like human object proposals as often as possible.

Beyond this, the stored and cropped images of people are compared to live footage through using a Euclidean color distance measurement. This allows for people to be labeled uniquely, serving as an object identification system as well. Each cropped image is analyzed in near real-time, pre-storage, to determine the six primary colors present on the person. The color content of their clothing, skin, and hair, identified by these most frequent colors, are compared to already stored data to see if a label can be assigned to them. If no match is found, a new label is made, as evidently a new person has been identified. This label system is beneficial as such identification aides in security, as cross-referencing the same person in different footage can lead to stronger analysis, and also can reduce the time needed to find specific people in a large database of footage.

The final implementation is divisible into the following general functionalities:

1. A canny edge detection method to find initial edges in a very raw form
2. A bit-masking algorithm that removes similarities between calibration image and live footage
3. Edge grouping that combines edges based on orientations through a flood-fill algorithm
4. Density-based clustering for finding bounding boxes of nearby edge groups and making proposals
5. K-means clustering functionality which finds the six most frequent colors in a proposal
6. Euclidean distance color comparison that compares stored and new object proposals' colors
7. Extracted image storage in a folder, alongside color data storage in a CSV for object labeling

2. Literature Review

2.1 Original References and Implementation

2.1.1 Edge Box Identification

The original sources for this solution proposed various techniques for generating object proposals, in addition to color differentiation. A paper by C. L. Zitnick and P. Dollár, titled “Edge boxes: Locating object proposals from edges,” employed significantly different late-stage methods compared to this project. They recommended scoring different edge groups based on affinity, then combining them further using a cosine similarity function. This approach would be applied after contour-removal algorithms to determine which edges correspond to specific objects in the image [2]. This method establishes an affinity threshold to decide whether two edges belong to the same object, using this scoring function:

$$a(s_i, s_j) = |\cos(\theta_i - \theta_{ij})\cos(\theta_j - \theta_{ij})|^\gamma \tag{1}$$

Furthermore, this function was derived for bounding box computation and the grouping of contours:

$$h_b = \frac{\sum_i w_b(s_i) m_i}{2(b_w + b_h)^\kappa} \tag{2}$$

This approach works well when processing a single image, but in the context of live security camera footage, it proved too complex and inaccurate for effectively identifying human subjects. The affinity-scoring method was tested in this project but failed to enclose object proposals with a single, unified outline. Various affinity score thresholds were tried, but none were successful. The main issue with this approach lay in the process of drawing bounding boxes, as the grouped edges still required a clustering algorithm, making the affinity-scoring method for creating groups unnecessary.

Earlier ideas in this paper were used however, especially in the application of using edge detection as a basis. This is shown in the implemented edge-based functions in this project, that retain the same idea of orientation-based edge grouping through an alternative approach.

2.1.2 Color Content Identification

The other reference, by M.-M. Cheng and colleagues, included in the original project proposal, focused on identifying an object's color content. The initial plan was to create a Gaussian Mixture Model (GMM) to combine similar and adjacent pixel colors into more uniform ones [3]. This would help reduce color variance within a single object proposal, making color identification easier. However, it was realized that this approach was essentially a clustering algorithm, and more computationally efficient clustering methods were available as alternatives to the complex GMM model.

2.2 General Changes to Original Literature

2.2.1 Edge Box Identification Upgrades

Instead of using affinity-scoring and a sliding window for object identification, our project implemented a flood-fill algorithm to group edges by their orientations. This is somewhat similar to the original paper, as it calculates gradients and forms contours. The algorithm starts at a pixel's location and orientation, then recursively groups neighboring pixels with similar edge orientations, creating coherent edge groups that approximate object boundaries without scanning the entire image. The rest of edge box identification will be discussed in the *Technical Description* section, as it involves DB clustering and so on.

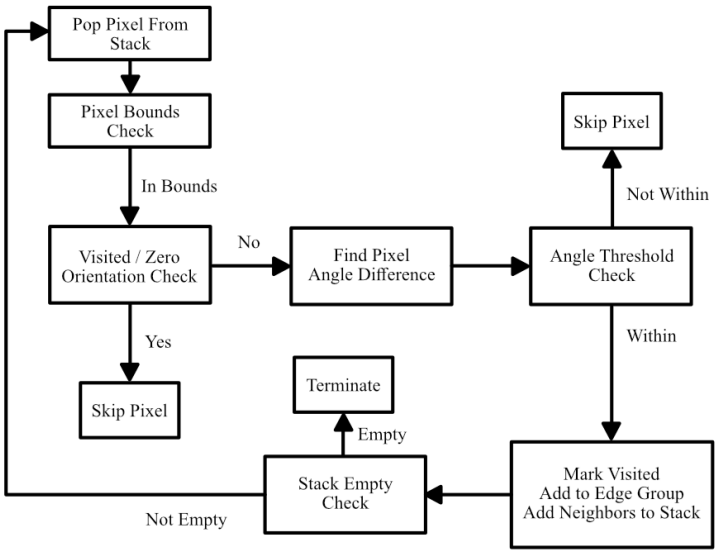

Figure 1. Custom flood-fill algorithm flowchart.

2.2.2 Color Content Identification Upgrades

To identify the color content of bounding box cropped images, we use Euclidean distance to compare an object's colors to stored color data. Instead of using a GMM, we apply a manually made K-means clustering function to reduce the object's colors to its ten most prominent clusters. The top six colors by frequency are then used for comparison, saving time in real-time computations by avoiding a more complex model. In the final Android implementation, background colors are removed during frequency calculations, ensuring that color identification focuses on the foreground.

Figure 2. K-means clustering applied to reduce color complexity and extract most frequent colors.

3. Technical Description

3.1 DSP Pipeline

The general DSP pipeline in our system processes camera feed data sequentially to detect, analyze, and categorize objects in real time. It begins with bitmasking via a "greenscreen" algorithm, which eliminates background information by comparing the current frame to a stored calibration frame. This generates a binary bitmask isolating foreground objects. To refine this mask, morphological operations such as OPEN and CLOSE are applied, reducing noise and ensuring a clean separation between the foreground and background.

Following this, edge detection is performed on the binary bitmask to generate a simplified outline of objects in the frame, leveraging canny edge detection for efficiency. The resulting edges are further analyzed through orientation calculations using the structure tensor and grouped into edge clusters via a flood-fill algorithm, significantly reducing the number of points needed for subsequent processing.

With the refined edge clusters, a density-based clustering algorithm inspired by DBSCAN is used to identify coherent groups of edges, which are then enclosed within bounding boxes. Bounding boxes smaller than a predefined threshold are ignored to focus on significant objects. Each valid bounding box is saved with a timestamp for playback functionality.

Afterward, color analysis is performed within each bounding box using a custom K-means clustering algorithm. This identifies dominant colors for object labeling and categorization, such as clothing or skin tones. The pipeline integrates seamlessly with the user interface, allowing calibration, live processing, and playback functionalities, ensuring an intuitive and efficient workflow for real-time object detection and analysis.

3.2 Pipeline Block Diagram

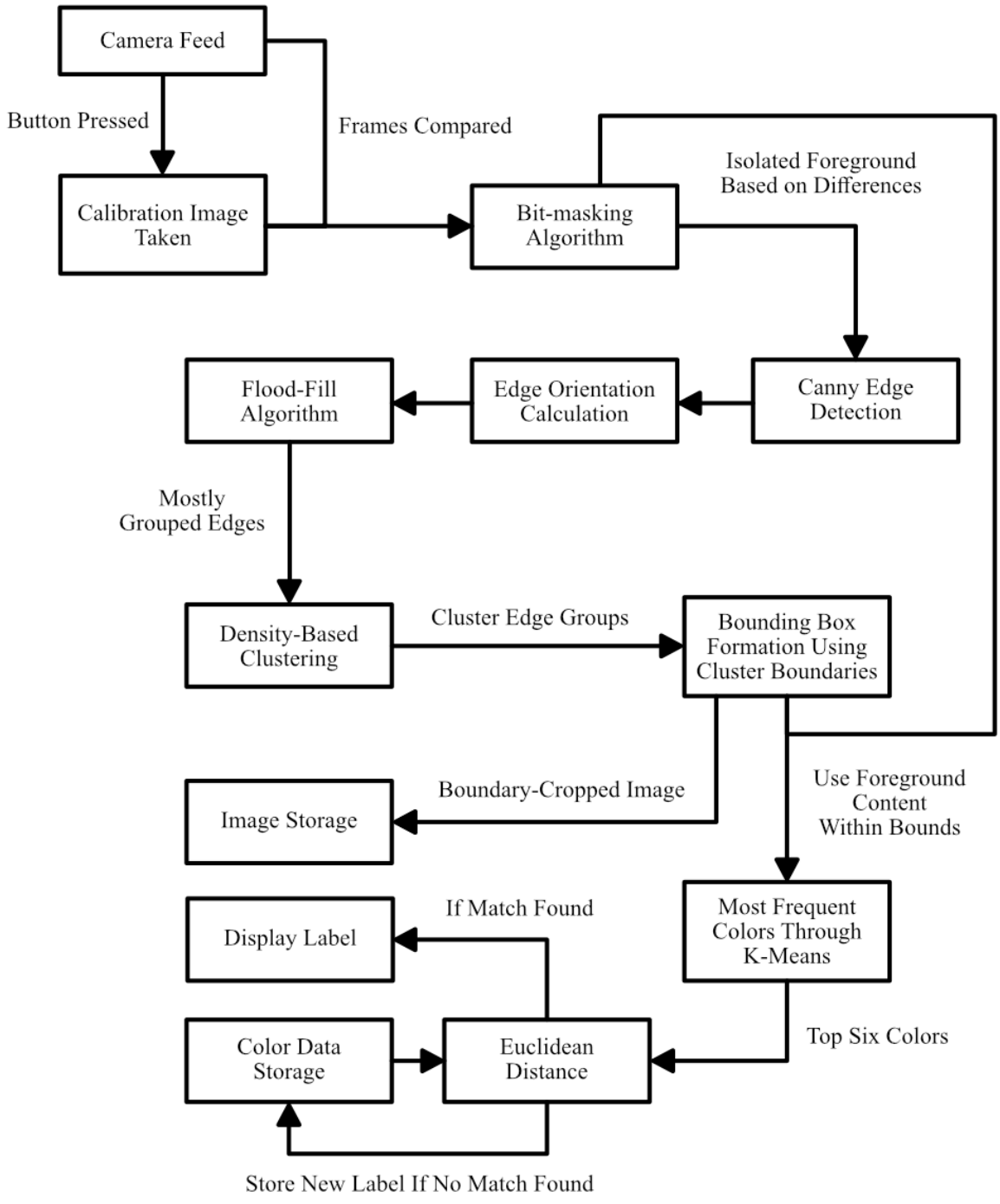

Figure 3. Full DSP pipeline block diagram from start to finish.

3.3 Bitmasking via “Greenscreen” Algorithm

3.3.1 Calibration Frames

We begin our processing by eliminating as much extraneous image information as possible. Given the context of a surveillance system, we make the assumption that we only need to work with objects in the foreground of the camera feed. To filter the background out, we first must know what the background looks like. We accomplish this by including a button to capture a “calibration frame”, which is intended to be the background of the camera feed with no humans present. By storing this frame, we can then compare it to the current camera feed and use the differences between them to identify objects which have entered the frame, creating an adaptive “greenscreen” based on any background, even ones with busy visual information. This frame can be reset to allow the user to change the camera’s position.

3.3.2 Bitmasking

Once a calibration frame has been captured, image processing can begin. Due to the high amount of ambient noise in the tablet’s camera sensor, we begin by applying a slight Gaussian blur to both the calibration frame and current camera frame in order to smooth out this noise. We then calculate the magnitude of the difference between each RGB pixel in the current camera frame and the calibration frame and use them to create a grayscale “difference image”. This image is then converted into a binary bitmask using a threshold which categorizes pixels below the threshold as “background” (0) and those above as “foreground” (1).

3.3.3 Cleaning Up

This thresholding can be made slightly inaccurate when either the object in frame is a similar color to the background, causing foreground to be miscategorized as background, or the tablet’s camera sensor automatically adjusts its light sensitivity in response to objects entering the frame, causing background to be miscategorized as foreground. To mitigate these inaccuracies, we apply OpenCV’s OPEN and CLOSE morphology operations to the raw bitmask to smooth it out. These operations are dependent on two other binary morphology operations, namely erosion and dilation. Erosion takes a binary matrix and radially shrinks all clusters of 1’s, “eroding” them away, while dilation expands these clusters, “dilating” them.

We first apply the OPEN operator, which equally erodes and then dilates the bitmask. The effect of this is that small patches of noise which were miscategorized as foreground are eroded away before dilating the mask back to its original size. The CLOSE operator is then applied, which closes small gaps in the foreground and reduces the presence of extraneous edges when edge detection is applied later on. After these operations have been applied, we are left with a relatively clean cutout of the foreground, effectively “greenscreening” the background away to simplify subsequent processing.

3.4 Edge Grouping via Flood Fill Algorithm

3.4.1 Edge Detection

One key realization we came to during design was that when generating a bounding box around an object, the outline of the object is far more important than the contents contained within it. To exploit this fact, we apply Canny edge detection directly to the binary bitmask we generate in section 3.3, rather than to the image which the bitmask contains. This generates a trace of the shape of objects which enter the frame, but contain minimal to no details of their texture. This differs from Zitnick's approach, which applied structured forest edge detection to the entire image to search for objects. These two optimizations significantly improved both the speed and accuracy of our implementation.

3.4.2 Edge Orientation Calculation

An outline of an object can still contain hundreds to thousands of pixels to recursively search and sort, which is unacceptable for a real-time use scenario. To further deconstruct the object to the bare minimum positional information, we reduce the outline to a set of relevant points along its boundary which still contain all the necessary information about its size and contour. To do this, we make use of the structure tensor, a matrix which encodes the gradients along the x and y axes of the image along with their correlations, and is given for each pixel as:

$$S = \overline{(\nabla I)(\nabla I)^T} = \begin{pmatrix} \overline{I_x^2} & \overline{I_x I_y} \\ \overline{I_x I_y} & \overline{I_y^2} \end{pmatrix} \quad \textbf{(3)}$$

We calculate these gradients using the Sobel filter. Eigenvalue and eigenvector decomposition of this matrix provides the magnitude and direction of maximum intensity change at each pixel. For our use case, the magnitude is irrelevant, so we calculate only the phase between the x and y gradients to generate an orientation map of the outline edges.

3.4.3 Flood Fill Grouping

Using this orientation map, we can then segment the outline by grouping pixels of a similar orientation into "edge groups". These edge groups are formed at characteristic points along the contour of the outline, and by decomposing each edge group to a single pixel at the edge group's average position, we can reduce the number of points which need to be searched by orders of magnitude. To do this we use a stack-based depth-first search flood-fill algorithm which searches along the outline edges. It traverses the outline, marking pixels as visited and adding them to an edge group as it progresses. We set a maximum angle deviation threshold which determines when one edge group is completed and a new one is formed, controlling the coarseness of the grouping algorithm. When this threshold is low, the algorithm generates many edge groups, and when it is high it generates fewer. We found that 45 degrees results in a good balance between accuracy and efficiency. Once edge groups have been formed, we calculate the average location of the pixels in each edge group, and keep only these centroids. By doing this, we reduce the number of pixels per object from thousands to usually less than one hundred.

3.5 Bounding Boxes via DBSCAN

3.5.1 Density Based Clustering

Utilizing the now averaged-out points for each edge group, we can generate clusters utilizing a manual density based clustering algorithm. This algorithm, based on a traditional DB-Scan algorithm, groups points that are closely packed together and separates noise points. It works by first identifying the neighbors of each point within a defined epsilon radius and checking if there are enough neighbors to form a cluster. If a point does not have enough points, it's marked as part of a noise set. Otherwise, the algorithm expands the cluster by recursively adding neighboring points that meet the density criteria. This process helps identify coherent and larger clusters of edges, which are then used to create object boundaries.

3.5.2 Bounding Box Creation and Saving

Now utilizing the edge clusters that are formed through the aforementioned manual clustering, we draw bounding boxes around the identified objects. For each cluster, we calculate the minimum and maximum x and y-coordinates to define the bounding box's position and dimensions. If the bounding box is smaller than 12x12 pixels, it is ignored and not counted as a significant enough cluster. For valid clusters, the image is extracted from within the box utilizing only its foreground. The image is saved for playback functionality presenting all masks in their sequential order. A timestamp is also generated over the live feedback to mark when the object was detected. The bounding box is then drawn with a green outline.

3.6 Color Analysis via K-Means Clustering Algorithm

3.6.1 General Color Analysis Reasoning

Color analysis on each bounding box and object proposal in the image is performed by k-means clustering each foreground-masked box into 10 color clusters. Utilizing the 6 most frequent colors in a bounding box, the identity of any object proposal is formed. This is used to gather information about a person's clothing, skin, hair, and so on.

3.6.2 K-Means Clustering Implementation

The manually-made k-means clustering implementation in this project is used to analyze the color content of each proposal. It begins by randomly selecting initial centroids from the pixel colors in the bounding box image. It then iteratively assigns each pixel to the nearest centroid and recalculates the centroids based on the average color of the assigned pixels. This repeats until the centroids no longer change. After clustering the pixels into 10 groups, the algorithm is complete.

3.6.3 Color Comparison and Object Labeling

Once the K-means algorithm finishes clustering, the six most frequent colors (the largest clusters) are used to generate an object label, which is either matched to an existing label based on color similarity or newly created and stored in a list. The color comparison is done using Euclidean distance, and if 3 of 6 current colors match those of stored color sets within a distance tolerance, the object is assigned the corresponding label. This enables the system to categorize objects based on their visual color patterns.

3.7 GUI Design

The GUI design of our project includes several primary buttons alongside a camera interface, as is pictured below:

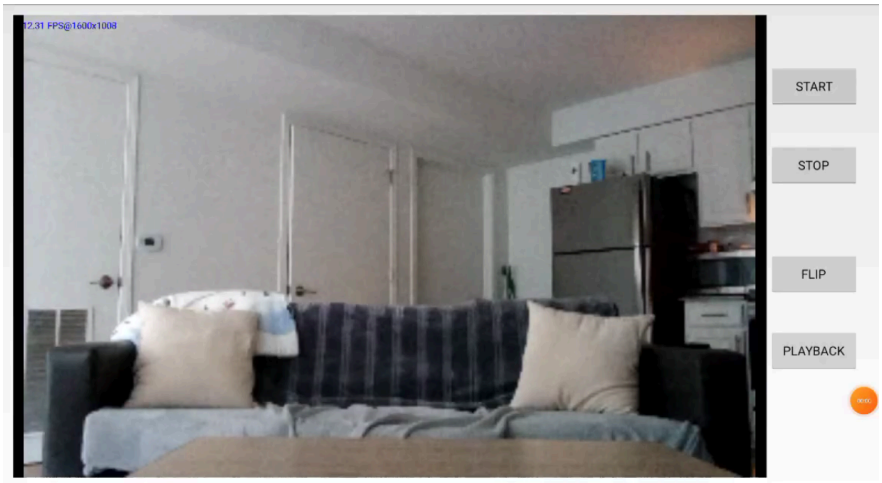

Figure 4. GUI design display of buttons alongside camera interface.

The “START” button is used to take a calibration image, which should be performed once when the tablet is at a standstill. Afterwards, the DSP pipeline initiates and begins from bitmasking, identifying object proposals, and so on. The “STOP” button halts the pipeline and live processing by removing the calibration image from storage. This allows the user to either flip the camera using the “FLIP” button or to enter the playback menu to view all stored objects via the “PLAYBACK” button.

Within the playback menu, the user may either clear the cache to remove all object images, or return the main camera interface menu with another button beneath the cache clearing one. They can also adjust a slider at the bottom to pan through the timeline of all images that are stored.

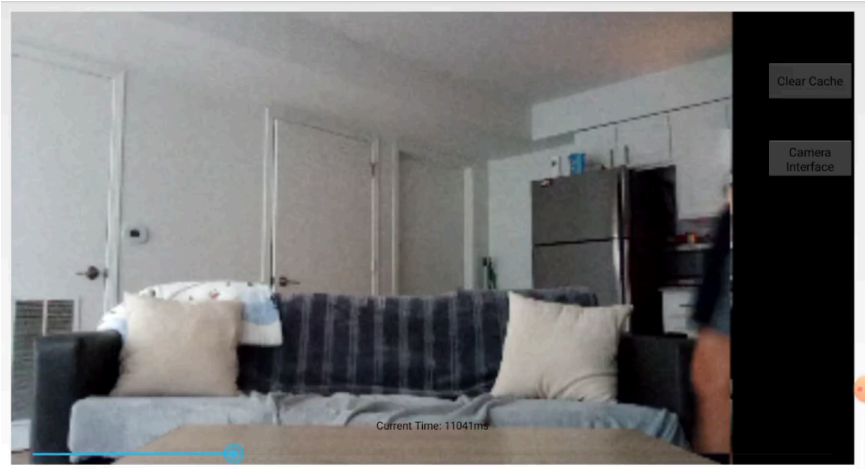

Figure 5. GUI design for playback menu.

4. Final Results & Screenshots

4.1 Results Overview

The final product proved to be accurate and comparable in terms of performance to the prototype model. This was primarily due to the optimization and minimization of edge groups by representing them via their average points, as is presented below:

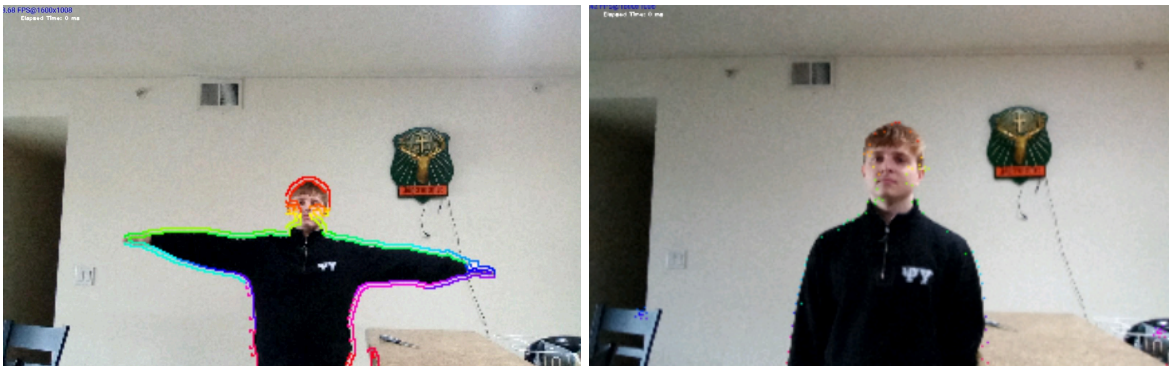

Figure 6. Reduction of multi-pixel edge groups into their average points.

This greatly enhanced the performance of our pipeline and allowed near real-time to persist in a slower environment such as an Android tablet. Furthermore, the general accuracy of each component of the pipeline was maintained strongly, and the identification of multiple objects in a single frame was improved in comparison to the prototype. This was accomplished as some parameters were tweaked in terms of the density based clustering algorithm to prevent cluster expansion from over-spreading.

Furthermore, the results of this project are quantifiable through drawing a ground-truth box and comparing its dimensional margins to that of the generated bounding box.

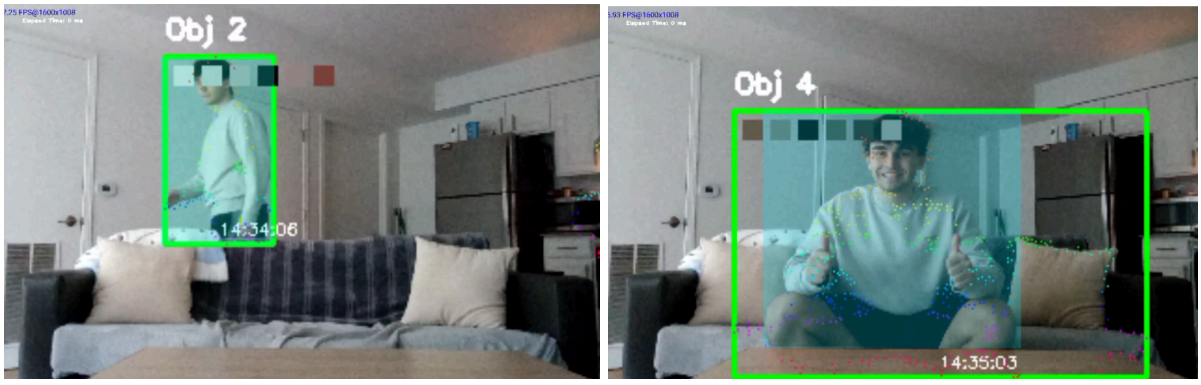

Figure 7. Strong performance example on the left, and poor performance on the right.

On the left-hand side, the ground-truth box only varied from the generated box by 5 pixels in the x-dimension and by 0 pixels in the y-dimension. Using a percent error function and its ground-truth width of 178 pixels, we compute $\left|\frac{173-178}{178}(100)\right|$ = 2.809% error. Equally weighing this with the 0% error in the y-dimension yields a total error of 1.405%.

On the right-hand side however, an anomaly occurred as the couch shifted since calibration, causing the pillow to be classified as part of the detected object. The x-dimension here varied 263 pixels from expected, and the y-dimension varied by 42 pixels. This amounts to a 36.990% error in the x-dimension and a 9.354% error in the y-dimension, averaging to a 23.172% total error.

Evidently, one part of the figure presents strong performance and the other side is a less common poor performance due to interference. Upon labeling and performing this sort of calculation process for 20 different images across one recording, we obtained an average total error of 4.819%. This is underneath the commonly used 5% threshold for statistical significance, and thus the results show that our pipeline is accurate enough to be considered reliable.

4.2 Testing Performed

As was planned in the original proposal for this project, some tests were performed to ensure general functionality of the final product. These consisted of testing various cases such as a single person standing still, a single person moving, and two people in frame as well. The first two test cases are visibly working as presented by the average error rate and figure in the previous section, but the third is yet to be shown. The following figure presents an example case of entering two different objects into frame:

Figure 8. Two objects placed into frame, showing multi-object detection.

Evidently, the detection of more than one object has been tested, and appears to be performing fairly strongly. The only deviation in the generated bounding boxes from the ground-truth boxes is due to shadow detection against the background pillow.

In terms of the y-dimensions, the card bounding boxes do not vary by any pixels from the ground-truth, but in regards to the x-dimensions they do. The left card varies by 7 pixels, whereas the right varies by 6 pixels. This results in an x-error of 11.666% in the left and 10% in the right, which becomes total errors of 5.833% in the left card and 5% in the right card when weighing in the y-errors. Considering this is a rare case where the drop shadow is directly behind the object, this performance is not bad, and visibly, it appears more than satisfactory.

4.3 Challenges Encountered

4.3.1 Accessing the OpenCV "ximgproc" library

Our initial approach used structured forest edge detection as opposed to Canny edge detection. We preferred this due to the comparatively cleaner edges it generates. This function is included in OpenCV's "ximgproc" extension, so we expected using it to be straightforward. However, the OpenCV package import used in this course, Quickbird Studio, turned out to not include this library. To get around this, we attempted to manually build the Shared Object (.so) files using CMake and add a link to them in the Quickbird Java Archive (.jar) file. Building OpenCV turned out to be a very tedious process, but we were able to successfully build the .so files.

However, we were unsuccessful in linking them in the .jar file, and Android Studio could not locate the "ximgproc" library. After several hours of failed attempts, we decided that structured edge detection was not worth the effort, and switched to Canny edge detection. However, this was before we decided to perform edge detection on the bitmask instead of the foreground image. We preferred structured edge detection over Canny because the latter generated very thin and sensitive edges, but it turned out that this behavior was well-suited to a binary mask. In the end, Canny edge detection turned out to be the better choice of algorithm.

4.3.2 DBSCAN speed

Throughout design, by far the largest processing bottleneck was the DBSCAN algorithm. Originally, we imported a library which performed clustering for us, as we assumed that manually implementing it was non-trivial and would be infeasible. This library ran quite slowly on top of the large number of points we were initially attempting to process. This resulted in a speed of up to 6 or 7 seconds per frame, which was unacceptable. This design is an exercise in minimalism, and we gradually increased our processing speed through continual reduction of the number of pixels we needed to represent an object.

Going from a bitmasked image to an outline to a handful of points representing the outline, we were able to increase our processing time to 1-2 seconds per frame. Finally, we decided to remove the dependence on an external DBSCAN library by manually implementing the algorithm ourselves. This final adjustment brought our speed to just under 0.5 seconds typically, which we were satisfied with.

5. Future Suggestions

5.1 Modifications

5.1.1 Bounding Box Merging

In regards to minor modifications that could've been made, we could have merged nearby bounding boxes either vertically, horizontally, or both ways. This was done vertically in the prototype model, and this helped performance and accuracy significantly. However, this was not done in this final Android model as the manual density based clustering function led to less fragmented bounding boxes than the DB-scan import in the original prototype did. Thus alternatively, the manual density based clustering could have been hyper-parameterized as well to optimize how bounding boxes are merged and separated.

5.1.2 Minor Playback Changes

Another minor set of modifications could have been the inclusion of timestamps and color compositions in the live playback menu. Currently, the final product displays the timestamp of images in the bottom via the slider, but a more clock-specific time could have been overlaid similarly to the live camera interface view. Additionally, the color swatches from each object's label data could have been displayed in this menu, or at least the object label itself.

5.2 Further Extensions

5.2.1 Supplementary Identification Methods

To improve object identification in real-time, additional techniques such as shape-based pattern matching could be implemented. These methods would complement our existing pipeline by allowing the system to recognize objects based on geometric features. For instance, combining the current color analysis with edge geometry comparison could help differentiate objects with similar colors but distinct shapes, maintaining a fast and efficient processing speed. This would also help resolve the issue of people wearing similar clothing, which caused them to be labeled as the same object. Minor changes as well, such as not allowing two bounding boxes to share the same object label could supplement this solution. This is assuming accurate bounding box merging is implemented first to prevent a single person from being misidentified as multiple objects.

5.2.2 Object KCF Tracking

Incorporating kernelized correlation filters (KCF) for object tracking would also enable our system to maintain consistent identification of objects as they move across frames. This method is lightweight and well-suited for real-time applications, leveraging the bounding boxes generated during the DSP pipeline to initialize trackers. By updating the object's position in each frame, KCF tracking could ensure continuity in detection, even in cases of partial occlusion or rapid movement, enhancing the system's ability to monitor dynamic environments.

6. Software & Hardware Documentation

6.1 File Responsibilities

The following five files are the core files utilized in this project:

1. ImageData.Java - Defines the image storage class for playback and image file management.
2. ImageProcessingUtils.Java - Contains image-processing related functions as part of the pipeline.
3. MainActivity.Java - Calls image processing functions and structures main program functionality.
4. Metadata.Java - Defines metadata class for storing timestamp and image dimensional data.
5. Playback.Java - Handles functionality of playback menu and image retrieval for display.

6.2 Functions Breakdown

6.2.1 ImageData.Java

1. clearCache - Deletes all files in the "ExtractedImages" directory within the app's storage and resets the global time variable to zero.

6.2.2 ImageProcessingUtils.Java

1. processGreenScreen - Subtracts a calibration image from an input image to create a binary mask highlighting changes between the two images.
2. applyCannyEdgeDetection - Applies Canny edge detection to an input image, detecting and highlighting edges using adjustable threshold values.
3. calculateOrientations - Computes gradient orientations for detected edges using Sobel operators and phase calculation.
4. groupEdges - Performs a depth-first search to group edge pixels with similar orientations within a specified threshold.
5. groupAllEdges - Iterates through an orientation matrix to group and find center points of edge groups across the entire image.
6. clusterDrawSaveAndColor - Clusters edge group center points, draws bounding boxes around clusters, saves extracted images, and performs color analysis.
7. formatTimestamp - Generates a formatted timestamp string in "HH:mm:ss" format.
8. saveImage - Saves an extracted image to the device's internal storage and records metadata about the image.
9. findOrCreateObjectLabel - Finds an existing object label with similar colors or creates a new one if no match is found.
10. getTopColors - Extracts the top N most frequent colors from an image using a custom K-means clustering algorithm.
11. performKMeans - Implements a K-means clustering algorithm to find color centroids in an image.
12. findClosestCentroid - Finds the closest centroid to a given pixel based on Euclidean color distance.
13. calculateMeanCentroid - Calculates the mean color of a cluster of pixels.
14. euclideanDistance - Calculates the Euclidean distance between two color points.
15. isSameScalar - Checks if two color scalars are exactly the same.
16. areColorsSimilar - Determines if two sets of colors are similar within a specified tolerance.

6.2.3 MainActivity.Java

1. onCreate() - Initializes the Android activity, sets up the camera view, permissions, UI elements, and configures button click listeners for calibration, stopping, playback, and camera flipping.
2. onStart() - Resets the timer when the user enters the camera interface.
3. onResume() - Enables the camera view and ensures OpenCV is properly initialized.
4. onPause() - Disables the camera view when the activity is paused.
5. onDestroy() - Releases camera view and output buffer resources when the activity is destroyed.
6. onRequestPermissionsResult() - Handles the result of camera permission request, logging the outcome.
7. onCameraViewStarted() - Initializes the output buffer when the camera view starts.
8. onCameraViewStopped() - Releases the output buffer when the camera view stops.
9. onCameraFrame() - Processes each camera frame by performing green screen detection, edge detection, edge grouping, and color visualization.
10. generateRainbowPalette() - Creates a list of color scalars with varying hues and transparency for visualizing edge groups.

6.2.4 Metadata.Java

1. Metadata() - Constructor that initializes a Metadata object with timestamp, coordinates, dimensions, and file path for an image.
2. getTimestamp() - Returns the timestamp of the image.
3. getTopX() - Returns the X-coordinate of the image's center.
4. getTopY() - Returns the Y-coordinate of the image's center.
5. getWidth() - Returns the width of the image.
6. getHeight() - Returns the height of the image.
7. getFilePath() - Returns the file path of the saved image.

6.2.5 Playback.Java

1. onCreate() - Initializes the playback activity, sets up the UI elements including a time slider, buttons for navigation and cache clearing, and configures the slider to filter and display images based on timestamp.
2. displayImages() - Loads and combines images from the metadata list into a single Mat based on their timestamps, creates a bitmap from the combined image, and displays it in the ImageView.

References

[1] P. Pellington, "How Much Data Does A WiFi Security Camera Use?" *Deep Sentinel*, Mar. 20, 2022. [Online]. https://www.deepsentinel.com/blogs/how-much-data-does-a-wifi-security-camera-use/. [Accessed: Nov. 19, 2024].

[2] C. L. Zitnick and P. Dollár, “Edge boxes: Locating object proposals from edges,” in European conference on computer
vision. Springer, 2014, pp. 391–405.

[3] M.-M. Cheng, G.-X. Zhang, N. J. Mitra, X. Huang, and S.-M. Hu, “Efficient salient region detection with soft image
abstraction,” in IEEE international conference on computer vision (ICCV). IEEE, 2013, pp. 1529–1536.