# Green Screen and Portrait Modes Using Foreground Segmentation on Real Time Video

Jason Vasko, Gustavo Fonseca

{jrvasko2, gvf3} @ illinois.edu

## I. Introduction

The goal of this project is to implement foreground and background segmentation of live video in an android application and to apply this algorithm to create effects such as a portrait mode with background blur and background replacement without a green screen.

The concepts of "foreground" and "background" in video are relatively easy to understand, but difficult to quantify for the purposes of automation. We will leverage the fact that in general, we expect the background to be more static and thus for background pixels to change less than foreground pixels. To achieve this, we will apply the algorithm described in the paper [1].

This project is unique in its applications of the concept of foreground-background segmentation, namely portrait mode and a green screen effect. Portrait mode refers to blurring the background of an image and keeping the foreground, usually a person, in an unaltered state. Green Screen effect refers to getting a replicating a the effect of green screen without the use of one, where the background would replaced with a user selected image and the foreground would remain unchaged.

## II. Literature Review

The main paper that we used to develop this algorithm was "Real-time adaptive foreground/background segmentation" [1]. This paper outlines the main steps of segmenting the foreground from the background. The paper outlines steps on how to create an efficient algorithm to limit computational time, which allowed the author to get it working in real time in small resolution videos [1]. The paper also outlines how to clean the result of the algorithm using various image

processing methods such as morphology functions [2] and a few computer vision algorithms implemented using the openCV library of functions.

The main difference our algorithm has to the one described in the paper is our use of the resulting segmentation. We will be using this output as a mask for two different modes: a green screen effect without a physical green screen, and a portrait mode similar to one found on a camera app. These mode will also be working in real-time on the app, allowing the user to see the effects of the segmentation as they happen.

Looking at other papers that used or improved upon the ideas presented in [1], one interesting paper I found focused on using this algorithm for surveillance purposes, such a making cities safer [3]. This paper details how this algorithm can be used to detect movement within live video streams, and thus allows the user to detect anomalies within very long security footage for instance. Another paper I found was using this algorithm in training a deep learning neural network [4], while this concept is very outside of the scope of this class, I found it interesting how applicable this algorithm is for video analysis.

## III. Technical Description

The main goal of this algorithm is to identify the foreground, or pixels changing within the video, and the background, or pixels that stay the same within the video. As new frames of video are processed we will compare each incoming pixel to past pixel values to determine if it is within the foreground or background. We will then use output of this segmentation to create the intended output effect.

Before the algorithm is used, the video we are using the algorithm on has go through a process known as **Cluster Representation**. The key idea behind this method is to create a group of $K$ clusters for each pixel. These clusters are past values of that pixel, thus each cluster represents a possible state that the pixel could be in the future. Each cluster has a weight $w_k$ and an average pixel value called a centroid $c_k$. For example, if a tree was swaying in the background of the video, the pixels representing that tree would oscillate between a few values. If there were no clusters, or no memory within the system, then the algorithm would recognize those pixels as changing and would classify them as within the foreground. With Cluster Representation we have the ability of remembering past values so those tree pixels could be classified as background pixels depending on the algorithm.

There are 4 main steps in this algorithm [1]:

1) **Cluster Matching**

For each incoming pixel, the algorithm matches the new pixel values to the nearest cluster. This distance is calculated by Manhattan Distance, as seen in Eq. 1. The main reason for using this distance instead of Euclidean distance is that it is much faster computationally as there is only addition and subtraction being performed rather than multiplication. Once we have calculated all the distances to each cluster, we compare the shortest distance to a predefined threshold $T$. If the distance is less than $T$ then this means that this pixel is likely a background pixel.

$$\text{Manhattan distance} = \sum |x - c_k| \tag{1}$$

2) **Adaptation**

This step of the algorithm updates the clusters and the weights associated with them.

If no match was found for the incoming pixel then the cluster with the smallest weight would be replaced by a new cluster with the incoming pixel as it's centroid and a predefined initial weight, (set to 0.01 in the paper).

If a match was found then we must update the weights of all clusters for that cluster group, seen in Eq. 2. Where $M_k$ is the cluster index of the matching cluster we found in step 1. $L$ is the inverse of the learning rate of the system, $\alpha$. We can change this number based on how quickly we want the system to adapt: small $L$ leads to large adaptation, while large $L$ leads to small adaptation.

$$w'_k = \begin{cases} w_k + \frac{1}{L}(1 - w_k), & \text{if } k = M_k, \\ w_k + \frac{1}{L}(0 - w_k), & \text{if } k \neq M_k, \end{cases} \tag{2}$$

We now need to update the centroid of the cluster. To avoid computationally expensive arithmetic and to not have fractional centroids, we can use an approximation of the equation given in Eq. 3. The approximation is as follows: if the error term $x_k - c_k$ is greater than $L - 1$, then the centroid will be incremented, and if it is less than $-L$, then the centroid will be decremented. This is much computationally cheaper than using Eq. 3 and avoids fractional centroids.

$$c'_k = c_k + \frac{1}{L}(x_t - c_k) \tag{3}$$

3) **Normalisation**

In this step we normalize all the weights of this cluster group to sum up to one as seen in Eq. 4, where $K$ is the total number of clusters. We can understand the weights of the each cluster as a probability of that pixel being that value when it is a background pixel. If the weight is high, then the likelihood of that pixel being used as the background is high, and vise versa.

$$w_k = \frac{w_k}{\sum_{j=1}^{K} w_j} \tag{4}$$

4) **Classification**

In the final step of the algorithm, we will sum all cluster weights that have a cluster weight higher than the matched cluster. The easiest way to do this is to sort the clusters by their weights, so our equation looks like Eq. 5. $P$ refers to the probability that the incoming pixel is within the foreground or background. We can then either make a binary decision based on some threshold or make a gray scale map of some sort to display what areas are more likely.

$$P = \sum_{k>M_k}^{K-1} w_k \tag{5}$$

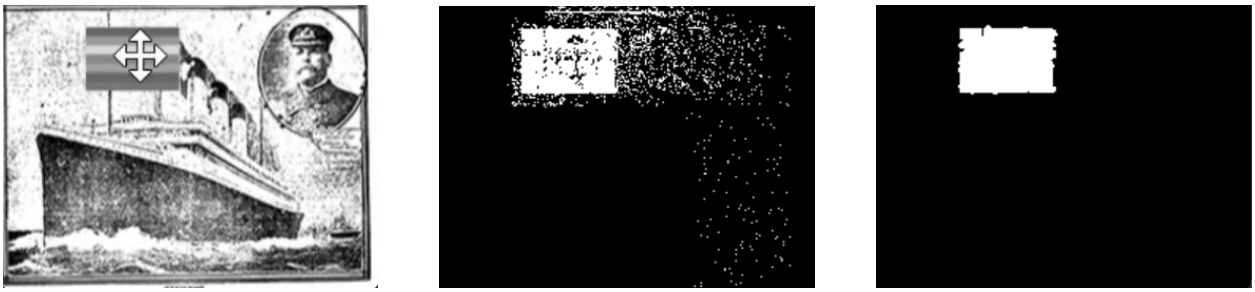
**Postprocessing:** Using this method alone will cause a large number of false positives and negatives in our results. Thus, we will use postprocessing to get rid of a lot of errors [1]. First, we will remove all lot of small false positives, stray pixels that were recognized as foreground, by using the open operation. The open operation is the combination of 2 image morphology operations, erode and then dilate [2]. The erode operation will look at an NxN area of a binary image, if all values in the binary image are 1, the erode output will be 1. The dilate operation will do the same procedure but instead if at least one pixel has a value of 1 in the area, then the dilate function will output a 1. An example of these operations can see can be seen below [2].

$$x = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \Rightarrow \quad \text{dilate}(x, S) = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{erode}(x, S) = 0 \quad (\text{all zeroes})$$

Here we are using a 3x3 square to do both operations.

The original algorithm would then use an openCV function to try to eliminate more noise. However, in the purpose of computational complexity and hardware processing power, we decided to implement every function in our final project by ourselves. This ultimately means that we had to exclude some of the post processing aspects that would make the algorithm too slow for real-time use. As a result, the only postprocessing we do on our app is the open function described above. The results of our postprocessing can be seen below can be seen in Figure 1 [1].



(a) Gray-scale Video Frame (b) Output before post-processing (c) Output after post-processing

Fig. 1: Output Stills for the different steps of the algorithm

**App Feature implementation:** Our app's goal is to use this segmentation produced by the main algorithm and add visual effects to a output image that the user can see in real-time. The program will use the segmentation as a mask for the two features, if the mask recognized a pixel as being in the foreground then the output pixel value will remain the same. If the segmentation sees it

is a background pixel then the program will check what mode the app is in. If in green screen mode, the original background pixel value will be replaced with the user's selected background image's pixel value. If the user is using portrait mode, then the background pixel will be replaced with a blurred version of that same pixel. The specific blur we used was a box blur of size 5, which takes the average value of the surrounding pixels in a 5x5 area. When in portrait mode, before any segmentation happens, the blur effect is applied to the entire image to be referred to after the segmentation during the portrait mode.

This algorithm allows us to get real-time video effects on low quality videos, with 320 by 240 being the highest resolution that produces a fluid video. Other resolutions such as 640 by 480 is more choppy with significant delay, around 5 frames a second.

**Flowchart:** See Figure 2 for entire procedure with the yellow steps highlighting the additional steps the program has to go through to implement our new features [1].

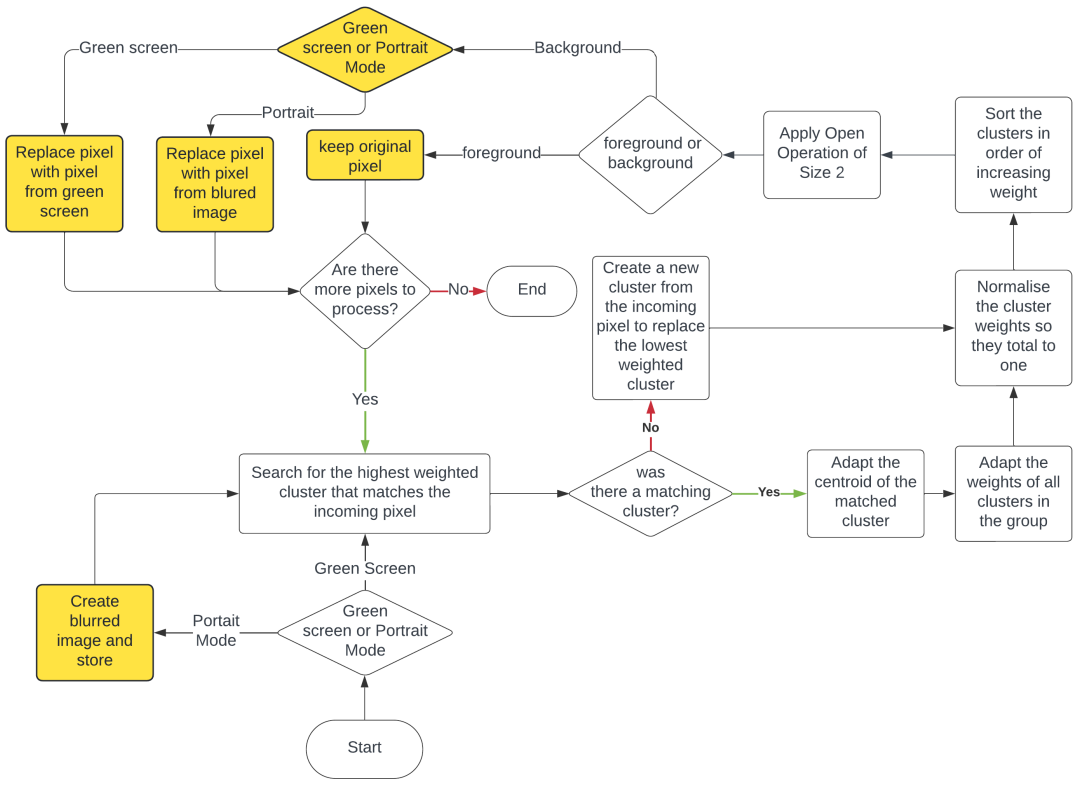



Fig. 2: Algorithm Flowchart

## IV. App Design Documentation

The software design of this project builds upon the base presented in lab 6. Figure 3 depicts the layout of the app's home screen and video feed screen. We have modified the home screen to allow the user to choose a background image using a spinner as a drop-down menu, and the chosen image is displayed using the ImageView on the home screen. In contrast, the video feed screen is almost identical to the screen used in lab 6 with different text. Note that the text "HistEq Image" is not actually shown in the app as it is always changed via code when the camera is started; this text is a relic from lab 6.

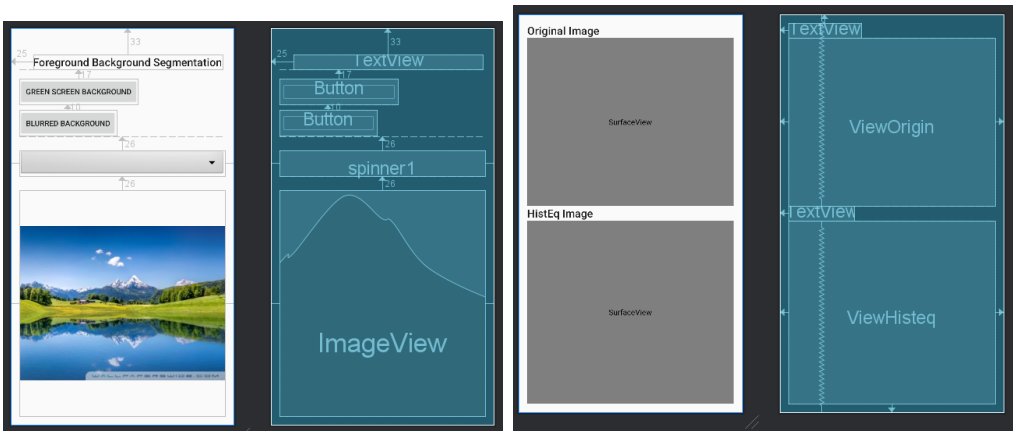



Fig. 3: App home (left) and video feed (right) layouts

Other than the xml files used to design the app layout, the only files used in developing this app are MainActivity.java and CameraActivity.java, which serve the same purposes here as they do in lab 6. For brevity, only modifications made from lab 6 will be discussed. First, MainActivity.java was modified to remove obselete references to buttons on the home screen that we chose to remove. We also added code to allow the user to choose an image using the spinner; this code is given in Figure 4. This code essentially stores the image labels in an array of strings and uses an ArrayAdapter data structure to describe how the spinner's options are displayed. We also define a function onItemSelected that runs when a choice is made; this function simply sets a flag to indicate which image was chosen and sets the ImageView on the app's home screen to display the correct image. Finally, we use the same method as lab 6 to allow the user to select between blurred background and green screen background, a public flag variable in the MainActivity class.

```
// get the spinner from the xml.
dropdown = (Spinner) findViewById(R.id.spinner1);
// create a list of items for the spinner.
String[] items = new String[]{"Lake", "Forest", "Beach"};
// create an adapter to describe how the items are displayed
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
    android.R.layout.simple_spinner_dropdown_item, items);
// set the spinners adapter to the previously created one.
dropdown.setAdapter(adapter);

// Add an OnItemSelectedListener to the spinner
dropdown.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
   @Override
   public void onItemSelected(AdapterView<?> parentView, View selectedItemView,
       int position, long id) {
      // Get the selected item from the spinner
      String selectedItem = items[position];

      // Check the selected item and set the flag variable accordingly
      if (selectedItem.equals("Lake")) {
         // Set flag variable to indicate choice 1
         imageFlag = 0;
         imView.setImageResource(R.drawable.lake);
      } else if (selectedItem.equals("Forest")) {
         // Set flag variable to indicate choice 2
         imageFlag = 1;
         imView.setImageResource(R.drawable.forest);
      } else if (selectedItem.equals("Beach")) {
         // Set flag variable to indicate choice three
         imageFlag = 2;
         imView.setImageResource(R.drawable.beach);
      }
   }

   @Override
   public void onNothingSelected(AdapterView<?> parentView) {
      // Do nothing if nothing is selected
   }
});
```

Fig. 4: Spinner code in MainActivity.java OnCreate method

```
class cluster {
   public int centroid;
   public float weight;
}
```

Fig. 5: Cluster class definition in CameraActivity.java

CameraActivity.java actually implements the algorithm. We use a simple class "cluster" to store the cluster data; this class definition is given in Figure 5. We define all of the relevant model parameters (learning rate, clusters per pixel, etc.) as private variables in the CameraActivity class.

The foreground/background segmentation algorithm itself is implemented in three main functions: `initClusters`, `matchUpdate`, and `classify`. Note that these functions simply implement the algorithms described above, so they are not included in this report for brevity. More specifically: `initClusters` simply creates new cluster instances for each pixel and sets the default centroid and weight values; it is called only once in CameraActivity.java's `OnCreate` method. `matchUpdate` performs steps 1, 2, and 3 of the algorithm (cluster matching, adaptation, and normalization) and is called once per incoming video frame in `onCameraFrame`. Likewise, `classify` performs step 4 of the algorithm and is also called once per incoming video frame. The postprocessing functions are implemented in another function, `postprocessing`, which uses helper function `applyMorphologicalOpen`, and `applyMorphologicalOpen` in turn uses helper functions `applyErosion` and `applyDilation` to perform the morphological erosion and dilation operations. The postprocessing function is also called once per camera frame. Note that we originally intended to use the morphological close operation in postprocessing as well, which motivated our decision to organize the code using so many helper functions, but we later found that using this function hindered the functionality of the app, so we removed it.

The remaining functionality of the app is implemented in the functions `greenmask` and `blurbg`. It is helpful to first examine the `onCameraFrame` method to see how everything fits together; this code is given in Figure 6. Here we can see the use of the flags set in MainActivity.java. In either case, the entire segmentation and postprocessing algorithm is ran, but in the case of a green screen background we run `greenmask` and in the case of a blurred

background we calculate the blurred background using `conv2`, the convolution algorithm from lab 6 modified to use a 5x5 box blur kernel instead of a 3x3 kernel, and we run `blurbg` instead.

The code for `greenmask` is given in Figure 7. Here, we set each output pixel to the input value if the mask from the segmentation algorithm indicates a foreground pixel, otherwise we retrieve the pixel value for the user's chosen image from a bitmap defined as a class variable. Note that this method required accessing the images from their location in the resources folder to initialize the bitmap in the `OnCreate` method; this code is given in Figure 8.

Lastly, the code for `blurbg` is given in Figure 9. This method is also quite simple; if the mask from the segmentation algorithm indicates a pixel is in the foreground, it sets the output value to the input value, otherwise it sets the output value to the value of the corresponding pixel in the blurred image calculated earlier.

The final notable code change is in the method `SurfaceCreated`, where we set the exposure offset of the camera and locked the exposure level. This improved the results of the app dramatically; see the results section for details.

## V. Results

Overall, our app's background blurring and green screen functions work as expected. Figures 10 and 11 show examples of the app's functionality. Figure 10 shows the object appearing over the background when moved in front of the camera; while there is some noise, it is clear that the object is displayed over the background. Figure 11 shows the blurring in action. The first image shows the blurring of the stationary background. The second image shows the result of moving a new object into frame; note the text on the box which is relatively sharp and legible. In contrast, consider the same object after leaving it in frame for some time. The segmentation algorithm used in our project detects movement to predict foreground and background pixels, so after some time has passed the algorithm considers the object to be part of the background and thus it is blurred as well. Note that the text on the box is no longer sharp or legible whatsoever.

We tested our app thoroughly in multiple settings with different objects and lighting levels. We found that the algorithm works best with objects that are farther from the camera or smaller, likely because these objects then have a smaller impact on the lighting of the rest of the area and therefore are unlikely to change any background pixel values enough to cause them to be mistaken for foreground pixels. We also found that the lighting has a significant impact on the

```
// Camera Preview Frame Callback Function
protected void onCameraFrame(Canvas canvas, byte[] data) {

   Matrix matrix = new Matrix();
   matrix.postRotate(90);
   int retData[];

   // Apply foreground/background segmentation
   if (MainActivity.convFlag == 0) {
      // Green screen background
      int[] locs = matchUpdate(data);
      int[] mask = classify(locs);
      int[] out = postprocessing(mask);
      int[] result = greenmask(out, data);
      retData = merge(result, result);
   } else {
      // Blurred background
      int[] blurred = conv2(data, width, height, boxBlur55);
      int[] locs = matchUpdate(data);
      int[] mask = classify(locs);
      int[] out = postprocessing(mask);
      int[] result = blurbg(out, data, blurred);
      retData = merge(result, result);
   }

   // Create ARGB Image, rotate and draw
   Bitmap bmp = Bitmap.createBitmap(retData, width, height,
       Bitmap.Config.ARGB_8888);
   bmp = Bitmap.createBitmap(bmp, 0, 0, bmp.getWidth(), bmp.getHeight(), matrix,
       true);
   canvas.drawBitmap(bmp, new Rect(0,0, height, width), new Rect(0,0,
       canvas.getWidth(), canvas.getHeight()),null);
}
```

Fig. 6: onCameraFrame method from CmaeraActivity.java

```
private int[] greenmask(int[] mask, byte[] data) {
   int[] result = new int[width * height];
   for (int i = 0; i < width * height; i++) {
      if (mask[i] != 0) {
         result[i] = data[i] & 0xFF;
      } else {
         // Set to the image
         int pixel = bm.getPixel(i / width, i % width);
         int alpha = Color.alpha(pixel);
         int red = Color.red(pixel);
         int green = Color.green(pixel);
         int blue = Color.blue(pixel);
         int gray = (int) (red * 0.299 + green * 0.587 + blue * 0.114);
         result[i] = gray;
      }
   }
   return result;
}
```

Fig. 7: greenmask method from CmaeraActivity.java

```
Resources res = getResources();
if (MainActivity.imageFlag == 0) {
   bm = BitmapFactory.decodeResource(res, R.drawable.lake);
} else if (MainActivity.imageFlag == 1) {
   bm = BitmapFactory.decodeResource(res, R.drawable.forest);
} else {
   bm = BitmapFactory.decodeResource(res, R.drawable.beach);
}
```

Fig. 8: Initializing the bitmap to access pixel data from the background images in OnCreate method in CameraActivity.java

```
private int[] blurbg(int[] mask, byte[] data, int[] blurred) {
   int[] result = new int[width * height];
   for (int i = 0; i < width * height; i++) {
      if (mask[i] != 0) {
         // Set to the original image
         result[i] = data[i] & 0xFF;
      } else {
         // Set to the blurred image
         result[i] = blurred[i];
      }
   }
   return result;
}
```

Fig. 9: blurbg method from CameraActivity.java

app's performance due to the camera's exposure settings, which are discussed below among the problems encountered in this project.

We encountered a few issues when testing this project. First, we noticed that moving objects in frame changed the values of all the pixels instead of just the pixels of the object, resulting in large swathes of foreground that should not have been detected when an object was moved in frame. This occurred due to the camera's automatic exposure adjustment, which essentially adjusts lighting levels in response to more or less total light reaching the camera. In our case, introducing a new object into a video changed the amount of light reaching the camera, causing this adjustment to take place and changing pixel values of background pixels. We fixed this by locking the exposure level when the camera surface is created. However, this is only a partial solution because now the camera is incapable of adjusting to the lighting of different spaces and if the exposure offset hard-coded in our app is off, the output can be nonexistent because the pixel values are all black or white due to the image being significantly over or under exposed. As a result, ambient lighting has a very important impact on the app's performance. We also tested our app initially with both morphological open and close operations included in postprocessing. We got outputs that appeared generally correct but with significant amounts of noise, suggesting an issue with postprocessing. We found through trial and error that the app functioned better


Fig. 10: App green screen functionality: object absent (left) vs present (right)

if the morphological close operation was not included in postprocessing, so we removed this function. Lastly, we originally calculated blurred pixel values as needed in `blurbg`, but this caused strange dark lines around shadows to appear in our output. We are not entirely sure why this bug occurred, but we were able to fix it by calculating the blurred pixel values separately for the entire image.

## VI. Extensions and Modifications

The results of the app development were very promising. The real-time segmentation allows for a lot user expression, one such feature would be changing the background during the green screen mode during the segmentation and not just before. Likewise, the portrait mode could have varying levels of blur or use different blurring effects that can be changed during the real-time segmentation.

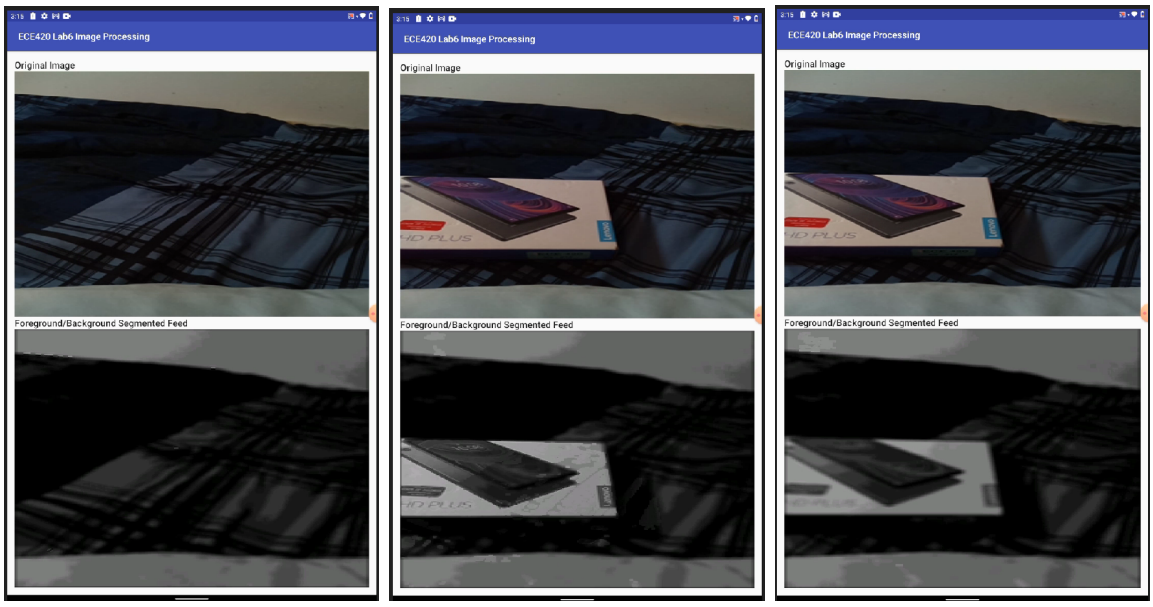



Fig. 11: App background blurring functionality: object absent (left) vs newly present (center) vs present after time (right)

Other modes could be added as well, such a zoom mode, where the real-time video would crop the video to only the area's that had a foreground pixel. This could potentially let users zoom in on a small moving object and get a better moving video than a manual zoom could.

Another aspect where the app could be improved would be incorporating color into its segmentation. Currently, the algorithm only takes the gray-scale intensity values into account in its segmentation, resulting in objects of similar intensity to the background being seen as background even if they are moving. The addition of this would make the segmentation much more accurate at the cost of higher computational complexity.

Some small quality of life improvements for the user could include: adding custom backgrounds from the user's photos, recording the effected video and saving it to the user device, and running the visual effects on user's saved videos instead only using the camera's live feed.

# REFERENCES

[1] D. E. Butler, V. M. Bove, and S. Sridharan, "Real-time adaptive foreground/background segmentation," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 14, pp. 1–13, 2005.

[2] P. Maragos, "3.3 - morphological filtering for image enhancement and feature detection," in *Handbook of Image and Video Processing (Second Edition)*, A. Bovik, Ed. Academic Press, 2005, pp. 135–156. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780121197926500723

[3] L. Sharma and N. Lohan, "Performance analysis of moving object detection using bgs techniques in visual surveillance," *International Journal of Spatio-Temporal Data Science*, vol. 1, no. 1, pp. 22–53, 2019.

[4] V. M. Mondéjar-Guerra, J. Rouco, J. Novo, and M. Ortega, "An end-to-end deep learning approach for simultaneous background modeling and subtraction." in *BMVC*, 2019, p. 266.