Video to Action Shot Sequence

By Aaron Knaack, Marissa Karp, and Elliot Shin

Website: https://actionsequence.wordpress.com

Abstract

Video to Action Shot Sequence is an algorithm using tools from the Vision System Toolbox provided by Matlab to develop an image from videos which represent an object's change over a period of time. The algorithm is designed to result in a composite image using several instances of a video and "stitching" the moving foreground object into one final image. Methods used to accomplish this can be summarized into a few categories: techniques such as Object Tracking, Foreground Detection using Gaussian Mixture Models (GMM), morphology operations provided by Matlab to clean masks, and processing the masks. The algorithm described in this paper has limitations for it's use such as the number of moving objects, the velocity of the object, and the morphology being used which depends on the overall quality and video attributes. Overall, the output from using the algorithm provides concrete results when the limitations are considered, and correct morphology operations are used.

Inspiration

Often times, photographers, ranging from novice to professional, put together action sequence of an object to demonstrate the movement in short time period and give dynamic effect to the photograph. This action sequence photograph is traditionally produced by manually editing multiple images using programs such as Photoshop. As much as the action sequence images look astonishing, they require a lot of effort and time. The goal of this project was to develop an algorithm to convert a video of an object in action into a single image that demonstrates the full action. This would mainly require three steps: align background while ignoring the moving object (foreground), detect the segment on the moving object, and cleanly put together all sequence images of an object into one background image.

The original goal for this project was to convert a dynamic video, with moving background, into action sequence image in MATLAB. This required particularly detailed background alignment due to movement of camera not allowing the sequence of images to share enough area to create clean background. One solution for this problem was to increase the amount of images used for creating the background. However, this was not the only issue for dynamic video conversion. When the input video is shaky or has rotation that creates low resolution frame, SIFT detection was not very reliable. Output image contained blurry and distorted image as a result. Thus, the project took a turn and the input video was limited to stable videos.

Method

I. Set Up Video

First, the desired video is retrieved and broken up into separate images, frame by frame. Each of these frames is then stored in an array. In this step, we also found information about the width and height of the frames as well as described a training variable used as a parameter for the Foreground Detector objects. It is important to note that adjusting this variable can influence the results significantly. Setting the variables early is good for consistency because they will be used several times in four separate Foreground Detector objects.

II. Get the Background

The next step is to find the composite background, filling all holes covered from the moving object. Stepping through the video, we stored the first frame (Figure 1) and last frame (Figure 2).

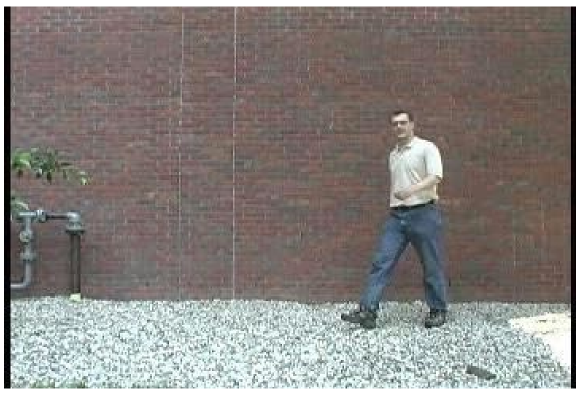


Figure 1

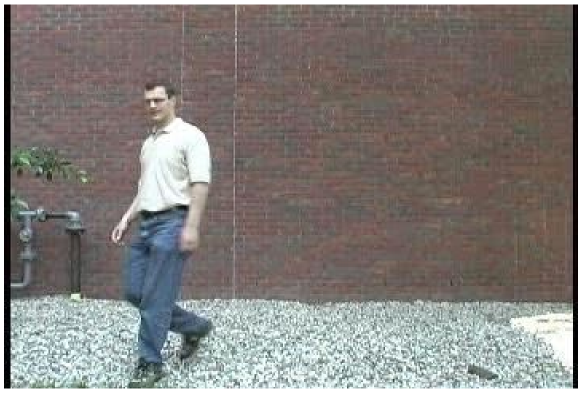


Figure 2

In combination with a method of background subtraction described by Chen, Stabler, and Stanley in their paper, "Automatic Generation of Action Sequence from Burst Shots" we used Foreground Detector which generated a mask of the last frame, and applied the difference of these two images in order to remove all foreground from the image (Figure 3 - 4). This created a final image with just the background of the video shown in Figure 5.

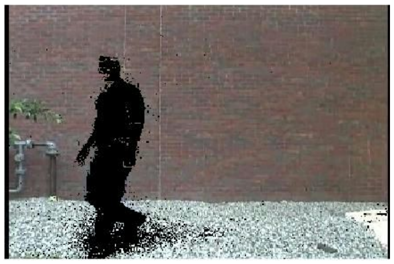


Figure 3

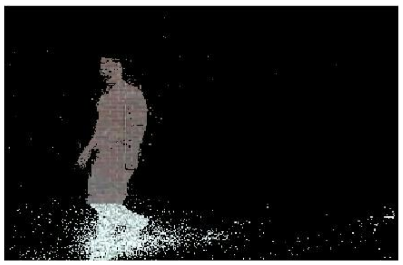


Figure 4

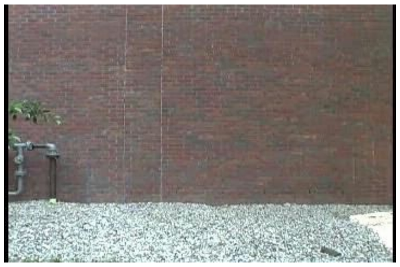


Figure 5

III. Determine bounds of moving object

Now that we found the background image, we needed to find the location of the moving object. Using background subtraction algorithm based on the Gaussian mixture models provided by the Foreground Detector tool in Matlab, detecting the moving object was made easy in concert with blob analysis, which detects groups of connected pixels. As seen in Figure 6 below, a bounding box is placed around the moving object. Due to the fact that blob analysis can detect several areas of one entire object, we made the assumption that the largest box bound in each frame *was* the moving object that we wanted to extract in it's entirety. Because of this, we stored the information for the largest bounding box of each frame. By using the centroids of the boxes we determined the object's direction using the average coordinate along the x-axis of the first 8 frames, and the last 8 frames. Currently, the program is restricted to left and right movement. Moving forward, we would strive to implement up, down, and diagonal movement capabilities.

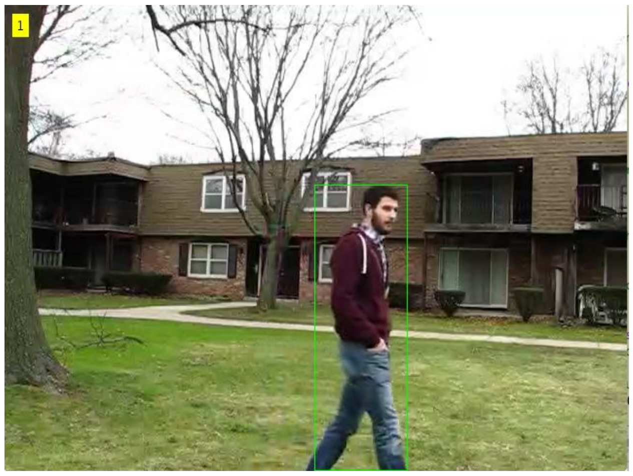


Figure 6

IV. Retrieve the frames where foreground doesn't overlap

An issue that we ran into during the trial process was that overlapping foreground objects from one frame to the next caused artifacts and failures with the output images due to rigid or incomplete masks, or oversaturated thresholds. Figure 7 and Figure 8 are both examples of failed output image due to overlap.

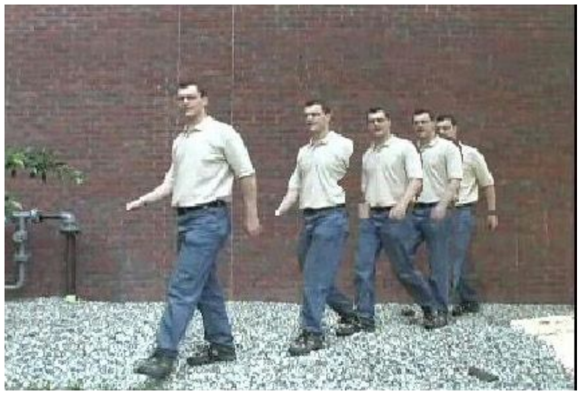


Figure 7

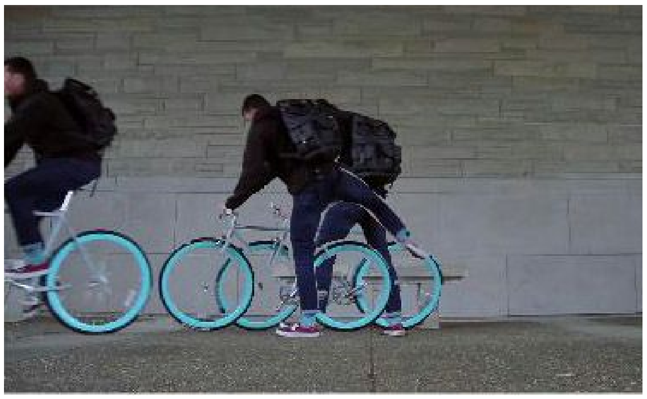


Figure 8

In order to avoid these issues, we implemented this step to determine which frames to include in the final output image based on where there was minimal to no overlap. A buffer was created that is dependent on the number of frames, the average bounding box area, and the video width. We then used that buffer along with each individual bounding box's width and x and y coordinates to determine whether or not those boxes are overlapping. If the separate foregrounds are not overlapping, the frame is considered a "non-overlapping frame" and used as a reference for the next iteration. These images are put into an array of "non-overlapping images." This array will be used as a list of which frames to stitch together in the final output image so that there is little to no overlap. The improvement from this buffer can be seen by comparing Figure 8 above with the new output in Figure 9 below.

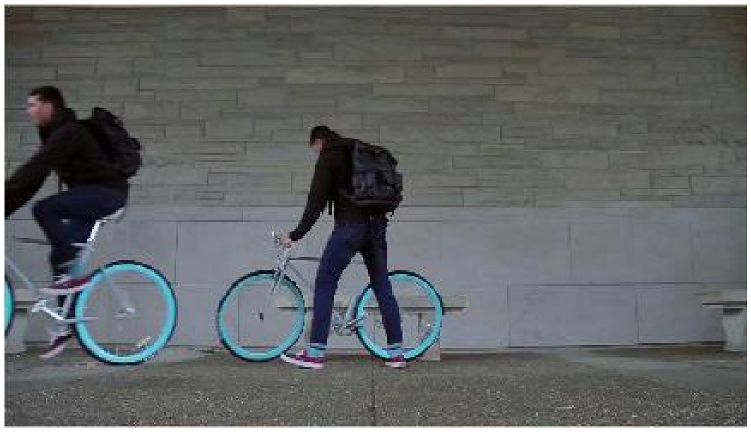


Figure 9

V. Create masks

Next, we need to extract the moving object from the chosen parts of the video. To do this, we needed to create a mask for each frame in the "non-overlapping image" array that was just created. These masks will block out any background pixels, erasing them in a sense. Reading through the video again and using the Foreground Detector, the foregrounds were then stored in a foreground masks array, but only taken from the second half of the video. This is because Foreground Detector used object tracking and predicts the path of the foreground object, and it's the prediction of this path which improves as the video proceeds. Only concerned with the better half, we needed to find a way to improve the masks from the first half. To accomplish this we reversed the array of frames stored at the beginning and created a new video which plays backwards. Repeating the last step we ran through that video again concerned again with only the last half of the video (ultimately the first half of the original video). Reversing the video and using the Foreground Detector once more allowed more accurate results for both halves of the video.

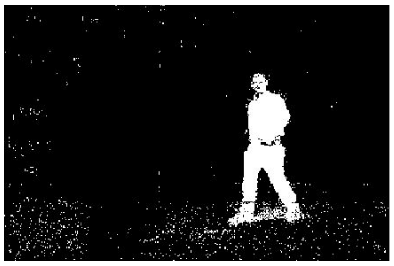


Figure 10

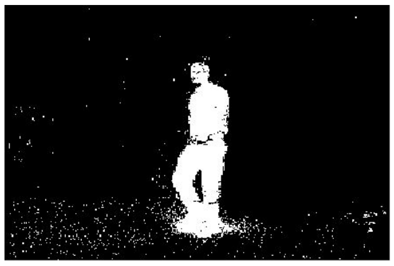


Figure 11

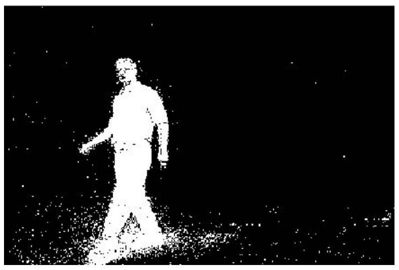


Figure 12

VI. Clean the masks

In order to get a more precise mask for each frame, we used morphology to clean up the masks. Combining the imopen, imclose, imdilate, and imerode functions with strel to clean the foreground object's mask, we can adjust for poor masks generated in the previous stage (Figure 10 - 12). Right now, the code used is dependent on the video, with different videos using different thresholds to create the best outcome. In the future, we would look into having user interaction with GUIs such as Shoelson's Image Morphology app that you can download through Matlab in order to determine the best threshold that should be used to create a mask appropriate for the video.

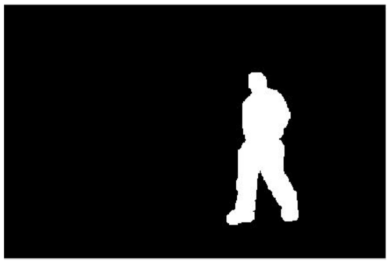


Figure 13

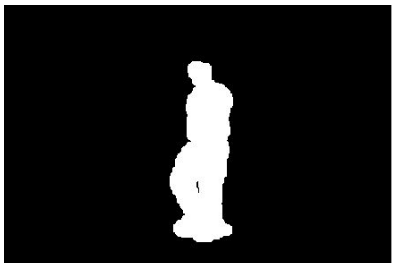


Figure 14

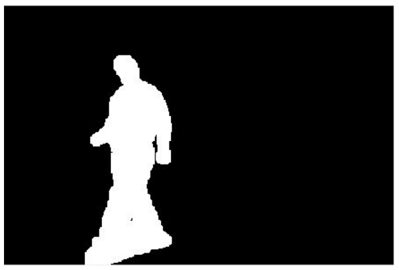


Figure 15

VII. Apply the Masks

The final task is to apply the masks to the background image to create the final product. We multiplied each foreground mask (Figure 13 - 15) and each corresponding foreground object image to create a cut out of the foreground object (Figure 17,20,23). Then, we applied the inverse of the foreground mask to the background (Figure 16,19,22). This creates a hole in the background images where the foreground object will be placed. Stitching together the foreground cutout to the background, pixel wise addition was done (Figure 18,21,24). This gives the final result image, which can be seen below, with each foreground object from the chosen images stitched onto the background image (Figure 24).

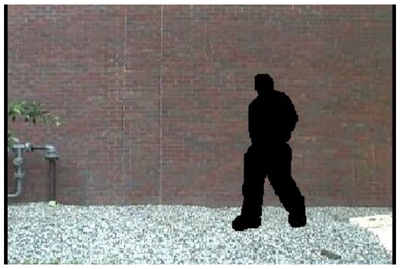


Figure 16

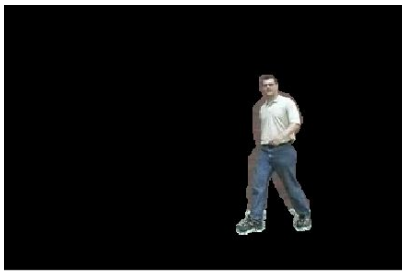


Figure 17

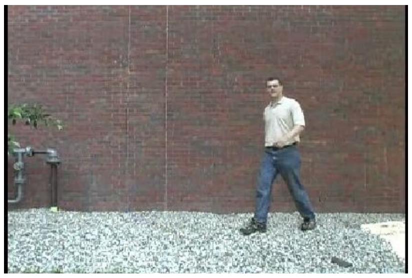


Figure 18

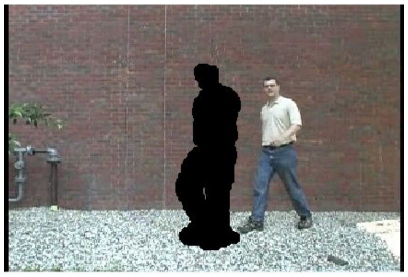


Figure 19

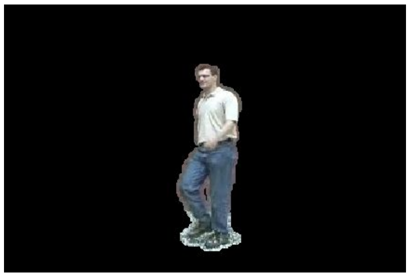


Figure 20

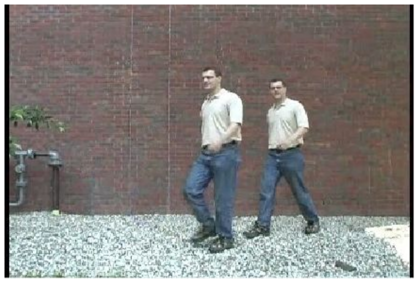


Figure 21

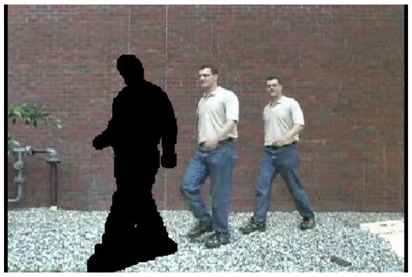


Figure 22

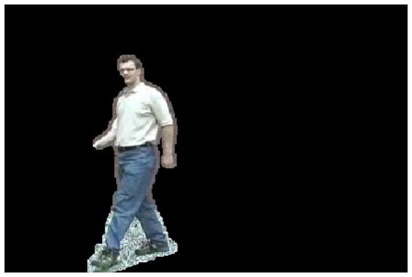


Figure 23

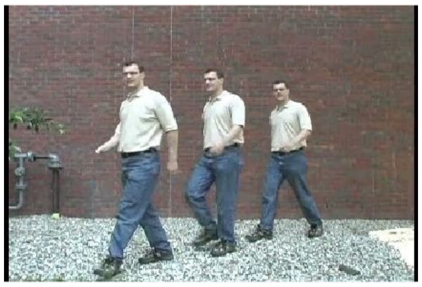
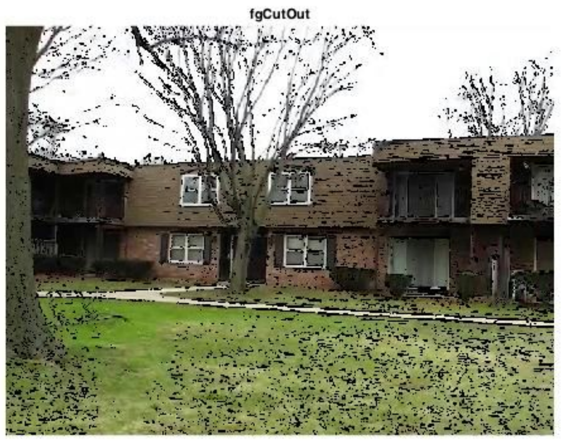


Figure 24

Limitations

There are a few very important variables to consider when using the algorithm. This is mostly stemmed from the type of video being used, and it's content. For example, because of the use of the Foreground Detector from Matlab's Vision Toolbox it is necessary that the video comes from a still camera. If the camera movement is dynamic along with the object of interest, the Foreground Detector cannot properly detect the foreground, and background subtraction cannot occur. However, small camera shifting during recording of the footage can still produce a fairly decent result, but the cost is a poorer quality output image with likely slight artifacts. This is mostly because small shifts create a mask that is slightly shifted from an adjacent frame, and produces a blurring or what I'll call "object extending" due to different pixels representing the same area be placed next to one another.



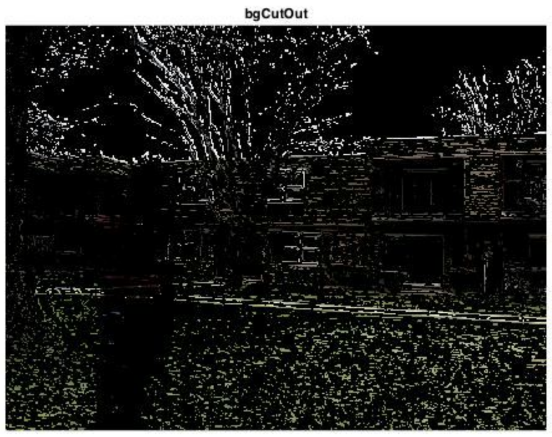


Figure 25 **Figure 26**

Notice in Figure 25 (the "fgCutOut") that much of the foreground detected was in actually the background caused by shifts in the background, and exactly the inverse cutout occurs in (the "bgCutOut") Figure 26.

Also, the algorithm has been written with some assumptions. That is: one foreground object is of interest, and another is that the object is moving either left or right. It should be noted that later implementations can improve the direction restriction. Ultimately, these limitations are a result of caution made to prevent overlapping foreground objects. We avoid using overlapping foreground objects because the masks can have impurities and imperfect edges, and *this* is a result of which Morphology Operations are used.

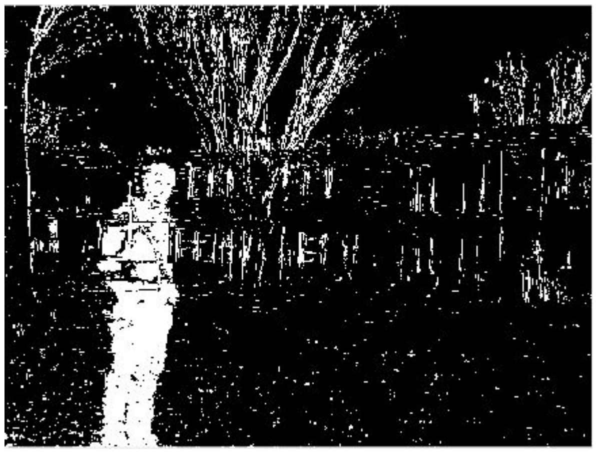


Figure 27

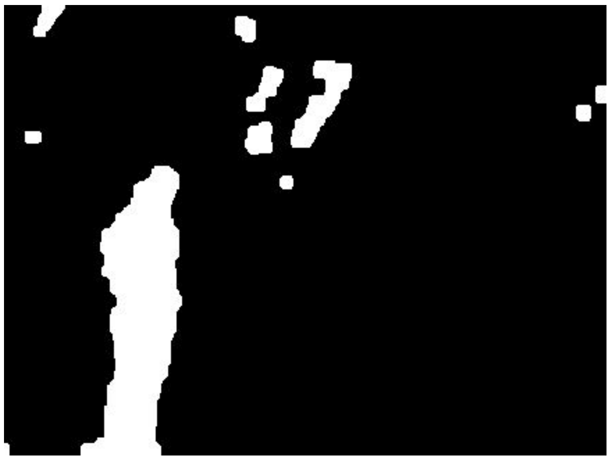


Figure 28

Figure 27 is before the morphology operations are applied, and Figure 28 is the resulting mask.

To ensure that the foreground objects do not overlap our algorithm uses object tracking and encloses the object with several bounding boxes. Once the objects are defined in each frame, it is assumed that the largest box is the most important and encloses most of the object of interest. The top left x,y-coordinates of the bounding box, including centroid, and width is used to compare adjacent frames and determine whether there is overlap of foreground objects. If there is no overlap of bounding boxes that frame is used next as a reference to be compared to in the following frames, and is stored as a non-overlapping frame to be used later. This is why determining whether the object is moving left or right is necessary - to compare correct coordinates of the adjacent frame's bounding boxes.

Another tool used to help ensure there is no overlap of objects is an arbitrary variable called the "buffer". The buffer adds a little additional spacing between foreground objects from the different frames. The buffer is used because much of the time the bounding boxes may not always provide accurate enclosures of the foreground object, especially at the beginning of the video. In Figure 29, the boxes are *of* the object, but there are many enclosures, where later in Figure 30 the enclosure is improved. Another example would be part of the tree being enclosed and not the foreground object at all.

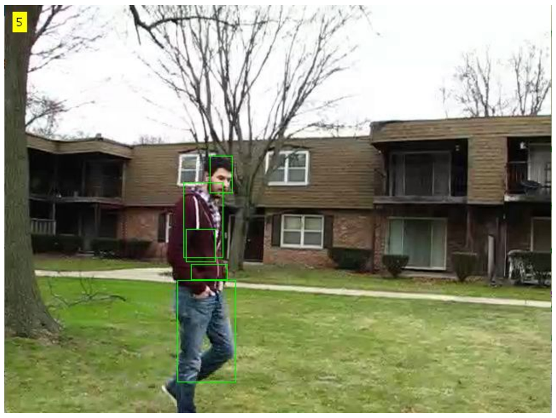


Figure 29

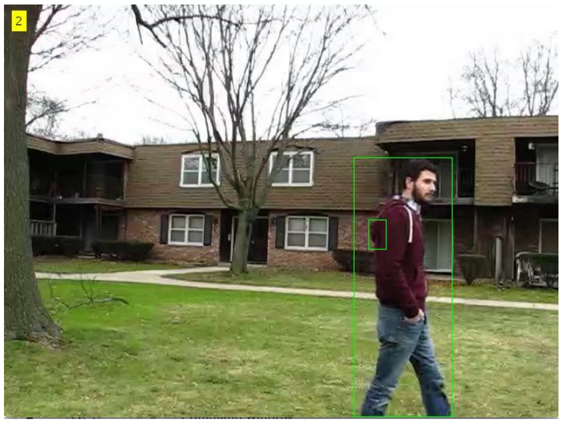


Figure 30

The buffer is calculated using the number of frames, and the width of the video. Adjusting the buffer manually can result in better output results, and finding a better buffer algorithm is one area that can most definitely be improved in. But, for now the buffer is calculated under the assumption that the foreground object is roughly the same height of the video and begins exactly on one side of the video and reaches the other side by the end of the clip.

Our algorithm is at it's best trying to accommodate for all several different types of video with these restrictions in mind. When each variable is considered and proper morphology can be used, the need to account for overlapping is not necessary. Understanding the algorithm in its completeness and understanding the variables aforementioned then the code may be edited to a specific input video and the results can be impressive. Here is a final example where more precise morphology was used specific to this video, and the overlapping restriction was emitted. Figure 31 is allowing overlap, but not using perfect morphology operations, whereas Figure 32 is allowing overlap and morphology has been improved to this specific video.

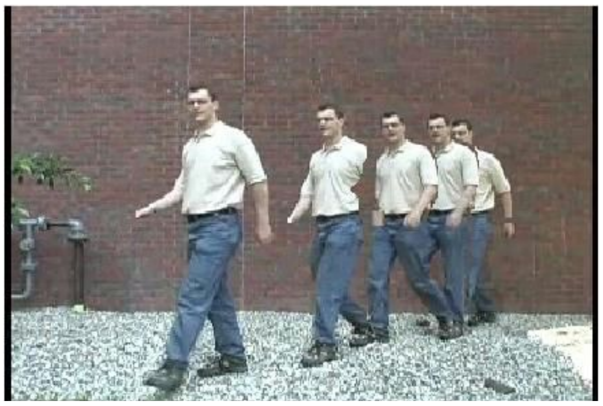


Figure 31

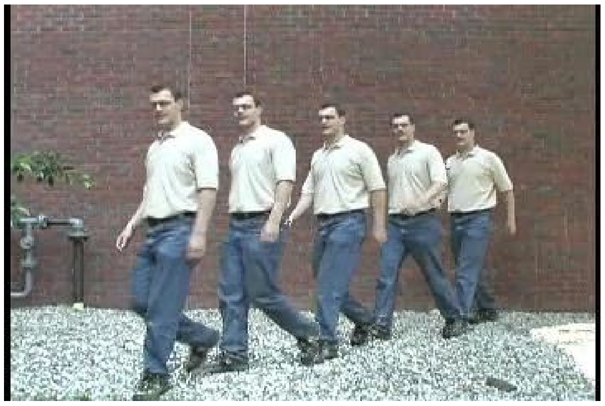


Figure 32

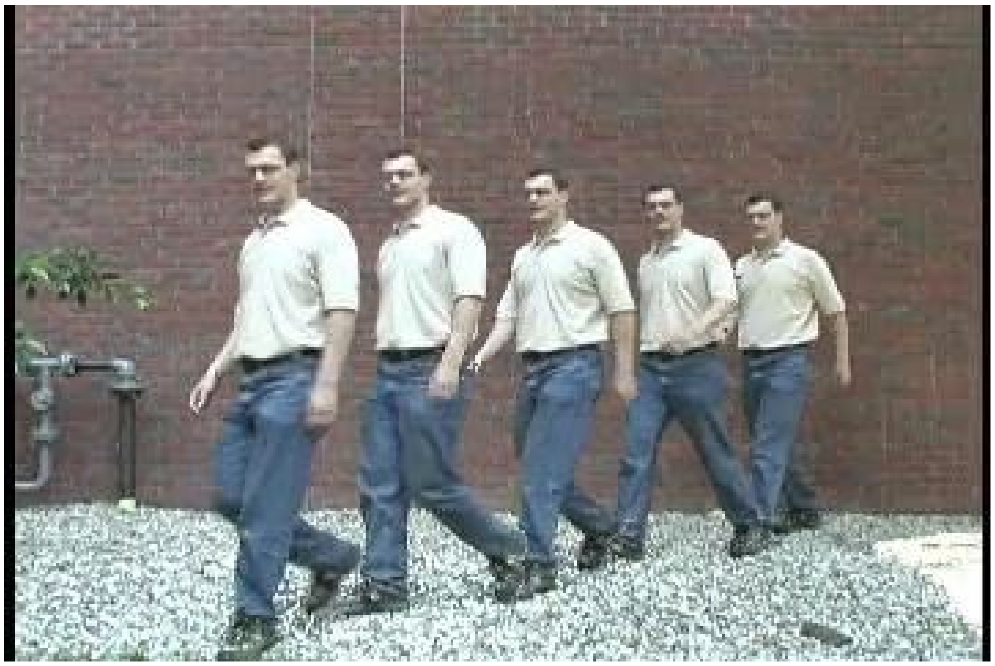
Job Partition & Useful Resources

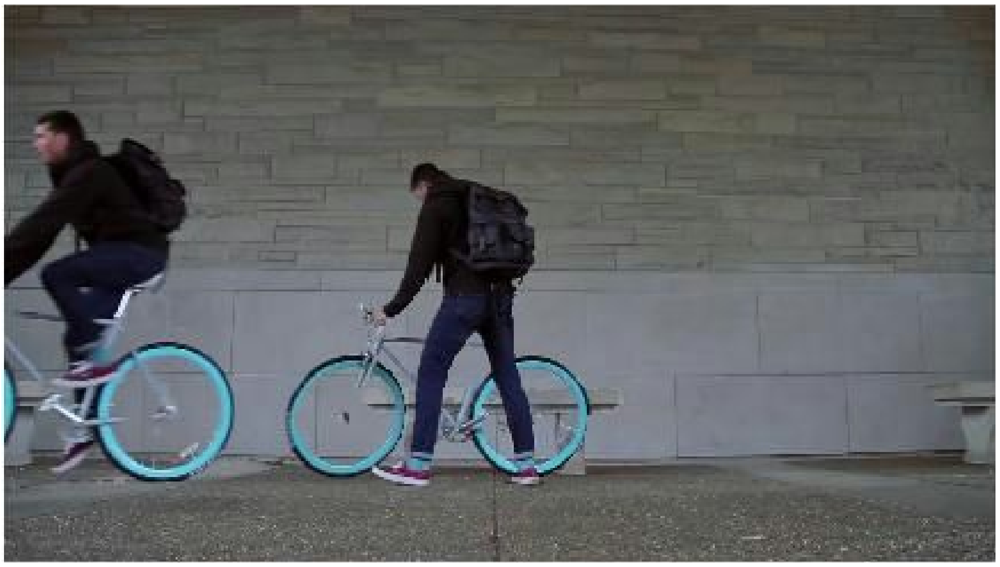
Elliot Shin : **Code:** Importing Video, Organizing Video Frames, Image Masking, **Paper:** "Inspiration"
Marissa Karp : **Code:** Image Masking, and Stitching Images, **Paper :** "Method", **Website**
Aaron Knaack : **Code:** Blob Analysis and Foreground Detection, Stitching Images, **Paper:** "Abstract" & "Limitations"

Chen, S., Stabler, B., & Stanley, A. (2013, June 5). Automatic Generation of Action Sequence Images from Burst Shots. Retrieved December 21, 2015, from https://stacks.stanford.edu/file/druid:yt916dh6570/Chen_Stabler_Stanley_Action_Sequence_Generation.pdf

Much of the example code found here was referred to while writing the code for our project: http://www.mathworks.com/help/vision/examples/detecting-cars-using-gaussian-mixture-models.html - more specifically in lines 110 - 143 of our code

Final Results





More Final Results