Final Project – Video Processing

Following the provided instructions, we divided the task into three subtasks. Firstly, we performed video stabilization to reduce unwanted camera motion. Next, we extracted the object of interest from the stabilized video. Finally, we utilized matting techniques to seamlessly integrate the extracted object into a new background, essentially "pasting" it in place. This sequential approach ensured a systematic workflow, allowing us to address each aspect of the task effectively.

The command to run the program is: python Code/main.py

When argument not provided the default video and background are being selected, to use a different one you can run:

python Code/main.py --input_video INPUT.mp4 --background_img PATH.jpg

Stabilization:

The objective of the stabilization module is to enhance video stability, enabling successful execution of the following operations.

Our approach for video stabilization used feature matching.
We iterate over the video frames and stabilize them with respect to the initial frame.

The stabilization process for each frame includes the following steps:

- **Extraction of keypoints and corresponding descriptors using SIFT**
  SIFT, or Scale Invariant Feature Transform, is an algorithm designed to identify salient key points and their corresponding descriptors in an image.
  It has the property of rotational invariance, which is vital for stabilization purposes.

  The workflow of SIFT involves the following stages:

  - Calculating octaves: The image is resized to multiple scales (octaves), with each octave containing a resized version of the image that is half the size of the previous octave.
  - Blurring: Within each octave, the image is convolved with a Gaussian kernel having different variances. Consequently, each octave consists of images with increasing levels of blur.
  - DoG: The DoG (Difference of Gaussians) is calculated by computing the difference between neighboring octaves.
  - Getting potential keypoints: Each pixel in the DoG pyramid is compared with its eight neighbors, along with nine pixels in the subsequent scale and nine pixels in the previous scale. If a pixel stands out as an extremum, it is considered a potential keypoint.
  - Selection of descriptive keypoints: Potential keypoints with responses below a given threshold and those representing edges are discarded, leaving only the most descriptive keypoints.
  - Assigning orientations: An orientation is assigned to each keypoint to enhance rotation invariance. The direction with the most prominent gradient is chosen as the keypoint's orientation, and subsequent calculations are performed relative to this orientation.
  - Keypoint descriptor: For each keypoint, a 16x16 pixel window is taken and divided to 16 sub windows of 4x4 pixels. Within each sub window, an 8-bin gradient histogram is computed to represent eight directions. The keypoint orientation is subtracted from each direction, and high values are saturated to handle illumination variations. These 16 histograms are concatenated to form a single 128-length feature vector, which serves as the keypoint descriptor.

- **Feature Matching Using a Brute Force Matcher**
  We employ a brute force matcher to compare descriptors between two frames. This matcher compares each descriptor in one image to all descriptors in the second image using L2 distance. As a result, every keypoint gets its best match from the other image. We used cv2.BFMatcher().match() for this step, omitting the crossCheck option due to stable results and timing considerations.

- **Finding an Optimal Homography with RANSAC**
  We used RANSAC (Random Sample Consensus) to find an optimal homography transformation that aligns the features.
  Homography, offering more degrees of freedom than an affine transformation, produced better results for our scenario.
  Our purpose was to find one homography for the entire frame, so we expected numerous outliers (for instance because of the person's movement).
  In order to do that we used RANSAC, known for its robustness against outliers.
  In our implementation, we used cv2.findHomography(...,method=cv2.RANSAC,...) for this step.

- **Warping With The Optimal Homography**
  Finally, we stabilize the video by warping each frame using the homography obtained between the frame and the initial frame. We used cv2.warpPerspective() for this purpose in our implementation.

The code implementing the stabilization part can be found in the file video_stabilization.py .

Background Subtraction:

The primary objective of the background subtraction block is to extract the person from the video. This block performs two key functionalities to achieve this goal. Firstly, it generates a binary video where the locations of the foreground, representing the moving person, are marked as 1, while the background is marked as 0. Secondly, it creates a video showcasing the extracted RGB image of the foreground, with the person, against a black background. The block takes two inputs: the stabilized video and the paths indicating the output locations.

The background subtraction process for each frame includes the following steps:

1. Fitting mixtures of gaussians – We created two mixtures of gaussians objects using cv2.createBackgroundSubtractorMOG2, one that fits on the original video (running forward) and the other fits on the video running backward. We are entering the frames of the video 5 times to the model to archive better separation between foreground and background.
2. Noise reduction preliminaries - To address the issue of noise, we employed two methods:
    a. KDE – We used sklearn.neighbors.KernelDensity to fit the pixels of the 5 first frames. Once the KDE is fitted, we predict the probability of a new point under the given PDF.
    b. Foreground Histogram – We converted the foreground pixels to grayscale and created a histogram. Then we calculate range of probable grey level that the object contains, it is done by converting the histogram to CDF and drop values of gray that the CDF is less then 0.002 or higher then 0.998 to reduce noise.
3. Extracting foreground – We start to create the binary mask in each frame:
    a. We are getting an initial mask to the frame from MOG, if the frame is in the first half of the video, we are using the forward MOG, otherwise the backward.
    b. Getting all connected components in the mask and leaving only the biggest one as foreground.
    c. Applying two morphological kernels. The first is 5x5 that reduce noise in the borders of the object. The second is 15x15 that fills holes in the object.
    d. We set zeros in pixels that are not in the intensity levels range that found in 2.b.
    e. We are refining the mask using KDE. In order to save time, we don't refine frames that outputs mask that is very similar to the pervious mask. After 5 frames without using KDE refinement, we are refitting the KDE.
    f. We are writing the mask to the binary video and multiply the original frame by the mask and writing it to the extracted video.

All relevant code for this block can be found in the background_subtraction.py file.

Matting and Tracking:

This block's goal is to perform matting and tracking.
The matting is of the image foreground with a new background image using a calculated alpha map.
The tracking includes detection of the moving object and plotting a bounding box around it.

The block contains implementation of the following steps:

- **Alpha Map Creation:**
  - We first create a trimap by subtracting morphological dilation from morphological erosion using 5x5 kernels. We then set the resulting area with a value of 0.5. We'll use the notation $B_\rho(\Delta)$ for this area.
  - We calculate geodesic distance maps from the foreground and background using the trimap's 0 and 1 values as initial foreground and background locations. We get two distance maps.
  - We calculate probability maps for the pixels to be in the foreground and background in the area $B_\rho(\Delta)$, using gaussian KDE on the luma part of the image. As a result, we get two probability maps: $\overline{P_f}(x), \overline{P_B}(x)$ for foreground and background respectively. For this part we use Scipy's gaussian_kde.
  - We combine the distance and probability maps to obtain alpha values for $B_\rho(\Delta)$ using the provided formula:

$$For\ every\ x \in B_\rho(\Delta):\quad \alpha(x) = \frac{W_f(x)}{W_f(x) + W_B(x)}$$

$$W_f(x) = \overline{D_f}(x)^{-r} \cdot \overline{P_f}(x),\ \ W_B(x) = \overline{D_B}(x)^{-r} \cdot \overline{P_B}(x)$$

Where $\overline{D_f}$ and $\overline{D_B}(x)$ are the foreground and background distance maps.

Note that this section is executed on a cropped area of the image to optimize computation time.
We extracted the bounding box using the trimap for effectiveness.

- **Matting:**
  We perform matting of the foreground image (human walking) with a new background using the alpha map.
  To do that, we follow these steps:
  - We set low and high confidences as "confident" foreground and background by thresholding the alpha map.
    The threshold values we used are 0.9 and 0.1.
  - We iterate over all "non confident" pixels in the alpha map and create a 7x7 area of interest around each pixel.
  - We minimize the equation:

$$\left|\left|\alpha(x) * C(X_F) + \left(1 - \alpha(x)\right) * C(X_B) - C(x)\right|\right|^2$$

For all possible combinations of X_F and X_B in the window.
$\alpha(x)$ represents the alpha value for the current pixel, $C(X_F)$ is the

foreground value in the original image within the window, and $C(X_B)$ is the background value in the original image within the window.
We Obtain the required foreground value for matting $(C(X_F^*))$ by finding the combination of foreground and background pixels that, along with the given alpha, produce the desired pixel value.

- We create the final image by adding the background for areas classified as "confident" background and the foreground for areas classified as "confident" foreground, and the values we calculated for non-confident areas.

- **Detection:**
  As stated before, we extract the detection area from the previously generated trimap. We convert the bounding box parameters to center format.

All relevant code for this block can be found in the matting.py file.

Running the code takes approximately 7 minutes on our computer.

Explanations of the CV2 functions we used in our code:

1. **cv2.connectedComponentsWithStats:**
   This function performs connected component analysis on a binary image and returns the connected components along with their statistics. It is useful for segmenting objects in an image and extracting information about each connected component, such as its area, centroid, and bounding box.
2. **cv2.getStructuringElement:**
   This function generates a structuring element that can be used for morphological operations such as dilation, erosion, opening, and closing. The structuring element defines the shape and size of the neighborhood used for these operations.
3. **cv2.morphologyEx:**
   This function applies a morphological operation to an image, such as dilation, erosion, opening, or closing. It takes the input image and the type of operation as parameters, along with the structuring element generated by cv2.getStructuringElement.
4. **cv2.createBackgroundSubtractorMOG2:**
   This function creates an instance of the Gaussian Mixture-based Background/Foreground Segmentation algorithm. It is commonly used for background subtraction in video processing to separate moving foreground objects from the static background.
5. **cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY):**
   This function converts an image from one color space to another. In this case, it converts a BGR (Blue-Green-Red) color image to grayscale, which reduces the image to a single channel representing the intensity values.
6. **cv2.boundingRect:**
   This function calculates the minimum bounding rectangle (x, y, width, height) for a set of points or a contour. It is often used to enclose objects or regions of interest with a rectangle.
7. **cv2.warpPerspective:**
   This function applies a perspective transformation to an image. It takes the input image and a transformation matrix as parameters and returns the transformed image.
8. **cv2.SIFT_create:**
   This function creates an instance of the Scale-Invariant Feature Transform (SIFT) algorithm. SIFT is used for detecting and describing local features in an image, which can be used for tasks like object recognition and image stitching.
9. **cv2.findHomography:**
   This function finds a perspective transformation between two sets of points. It is commonly used in computer vision tasks such as image alignment, stitching, and augmented reality.
10. **cv2.dilate:**
    This function performs morphological dilation on a binary image. It expands

the boundaries of foreground regions or objects by adding pixels to their edges.

11. **cv2.erode:**
    This function performs morphological erosion on a binary image. It shrinks the boundaries of foreground regions or objects by removing pixels from their edges.
12. **cv2.resize:**
    This function resizes an image to a specified size or scale. It takes the input image and the desired output size as parameters and returns the resized image.
13. **cv2.rectangle:**
    This function draws a rectangle on an image. It takes the image, the coordinates of the top-left and bottom-right corners of the rectangle, the color, and the thickness as parameters. It is commonly used to highlight or mark regions of interest in an image.