Ingegneria Informatica

Physical Distancing Detector

using the OpenCV and OpenPose Python libraries

Image and Video Analysis

Professor:
Pietro Pala

Student:
Paula Mihalcea

June - July 2020

Index

Abstract

This paper presents a Python program that, through the use of the OpenCV[2] and OpenPose[12] libraries, is able to process a video stream and detect if any of the people present in each frame are violating the physical distancing safety measures imposed by the 2020 COVID-19 pandemic. Although the OpenPose library is rather resource demanding (thus cannot be executed on any device), the software has been built with a real-world application in mind, and written with many practical measures meant to ease its setup and use for the average user.

1 Introduction

The year 2020 will certainly be remembered for the spread of COVID-19, a virus which gravely affected both the global economy and our daily lives. From the lockdowns that many countries imposed to their citizens to the mandatory use of masks, many measures have been taken in order to contain its rapid transmission; among these efforts we can find various regulations that require people to keep a minimum distance of about 1.8 – 2.00 meters among them, particularly in public areas.

The problem with the last measure is that, unlike the others, there is no easy way to correctly carry it out without having at least an idea about how long two meters are. It might seem trivial, but not everyone has a good sense of measure, and estimating the correct distance to keep from others might prove quite difficult for some.

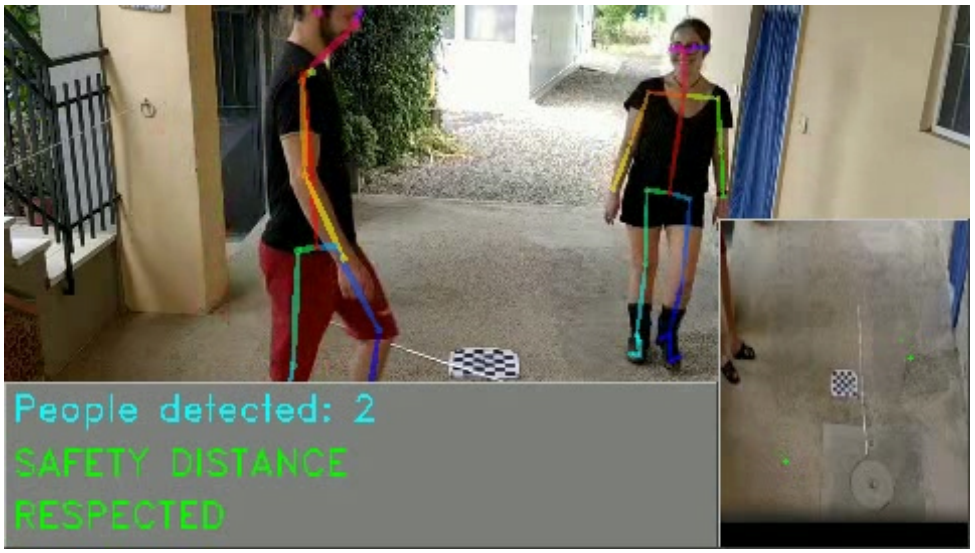

Figure 1: Two people that are respecting the safety distance.

Figure 2: The same people from fig. 1 are not respecting that distance anymore.

The idea behind the software presented in this paper is to offer a tool that can automatically measure the distance between two or more people and warn them if they are too close. It also shows a map that displays their position, providing said people with a way to better understand how much two meters (or any other safety distance is provided to the program) actually measure.

2 Operating principles

The program simply takes an input video stream (webcam or file) and overlays on every frame a status bar which informs about the number of people detected in said frame and whether they are respecting or not a given safety physical distance[1]. Along with this information is displayed a (partial) view from above of the visible floor, on which the estimated position of the detected people is marked with colored dots. An external `INI` file can be used to set up and customize almost every aspect of the program, from the appearance of the status bar to the dimensions of the map in pixels and meters.

2.1 Information needed

Depending on the information available about the environment framed in the video, the program needs different data to be able to properly create the map.

[1]From now on this distance will always be assumed to be equal to 2 meters.

In the worst-case scenario, it only needs the length of the sides of a rectangular mat that can be laid on the floor framed in the video and used to determine the dimensions of the area around it. Otherwise, if such a mat is not available, the user will have to specify at least the width and height of the visible portion of floor (or the desired part that is going to be selected).

2.1.1 Source points (corners)

In addition to this, since the geometric transformations used to create the map need at least four points in the video to work (the corners[2] of the visible floor or the mat), the user might also want to specify them in pixel coordinates for better precision; this is especially true in low resolution videos where a few pixels can make the difference between a good and a bad transformation (see figures 3 and 4).

Figure 3: Map created with mat source points carefully extracted and saved in the setup file.

Figure 4: Map created with hand-picked mat source points; it is visibly less accurate.

However, if these coordinates cannot be obtained prior to initialization, the program will prompt the user with the first frame extracted from the video and ask him or her to click directly on the image to determine them automatically. Technically, this method is as effective as specifying the points in the setup file, but in practice it is more prone to error as selecting the corners with a mouse is usually less precise than writing the actual accurate coordinates.

2.2 Assumptions

An important assumption needed for the program to work correctly is that the area framed by the camera does not change during execution (for example by moving the camera), as the map is created during the initialization phase and cannot be edited afterwards, if not by stopping the program and initializing it again.

[2]These corners will also interchangeably be called *points* from now on.

2.3 Workflow

After starting the program with a valid input source, which can be done by writing in a terminal opened in the main folder (`Physical distancing detector`) the following command[3]:

```
python3 physical_distancing_detector.py source [save] [
    destination] [setup_file]
```

the main script will check if the setup file already contains any reference corners in the form of pixel coordinates.

2.3.1 Corners detection

The possibility to specify these corners directly in the `INI` file was meant to ease the setup for those users that have a fixed camera and possess (almost) complete knowledge about the environment it has been installed in, in addition to improve accuracy. If corners are specified for both a mat and the visible floor, the program will ask the user which ones it should use to generate the map. Otherwise, as mentioned above, for those who do not know said points, it will prompt a window containing the first frame of the video, in order to allow the user to choose the corners by clicking on them then pressing `ENTER` (fig. 5).

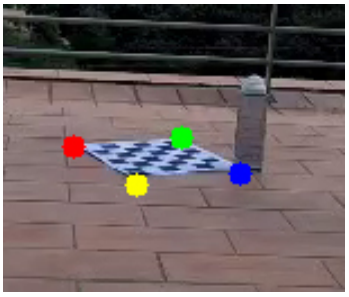

Figure 5: Mat corners selection window with all four points selected.

Additional borders

It is worth mentioning that if the user chooses to manually select the floor corners the program gives him or her the possibility to add a border to the image, in order to be able to generate a more accurate map by also selecting points that might fall outside the visible frame (see figures 6 and 7). This option is available by simply pressing `SPACEBAR` while in floor corners selection mode, then entering the desired thickness of the borders in pixels (fig. 8).

[3]See chapter 3.1 for details about the command line arguments.

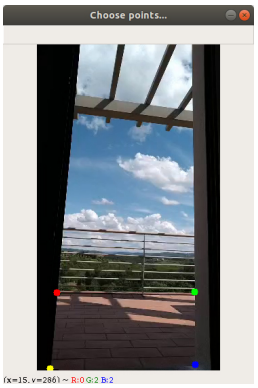

Figure 6: Source points selection window without additional borders.

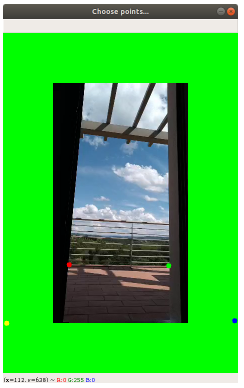

Figure 7: Source points selection window with 100px green borders.

```
Insert left border thickness in pixels: 100
Insert top border thickness in pixels: 100
Insert right border thickness in pixels: 100
Insert bottom border thickness in pixels: 100

Click on the four corners of the floor plane (top left, top right, bottom right, bottom left) then press ENTER,
or press SPACEBAR to add or change borders.
Otherwise, press ESC to exit.
```

Figure 8: Insertion of border thickness in the terminal.

2.3.2 Map generation

After getting the four corners (in a way or another), the program proceeds with the map generation by applying a homography to the quadrangle obtained by connecting the points. The four destination points for the homography (see 3.6) can either be determined automatically (as in the case of the mat, or of the floor if no other points are available) or also specified by the user in the setup file (again, this usually yields a better map in the floor setup, particularly for irregular views, as shown in figures 9 and 10).

Figure 9: Map created with automatically calculated destination points; perspective clearly isn't perfect.

Figure 10: Map created using user-specified destination points; perspective looks way better.

The map and the homography matrix generated during this step are saved for later use: the map for the video overlay, the matrix for determining the position of the people detected in the video.

2.3.3 Video processing

The program thus proceeds with feeding each video frame to the OpenPose library, which analyses it and returns the keypoints of the people it detects in it. By elaborating the heel keypoints and applying the homography matrix to their coordinates, the program determines the position of a person and prints it on the map, while also informing the user about possible violations of the physical safety distance.

From this point on, the program runs indefinitely or until the source video file reaches the end; it can be safely closed by pressing `ESC`.

3 Technical details

The program, written in the Python programming language, makes use of mainly two libraries: OpenCV for the video processing and map creation and OpenPose for detecting any people present in a given scene.

It is made of many components, of which the `physical_distancing_detector.py` script is the main one. In this section they will be analyzed one by one in order of appearance and somewhat importance.

3.1 Main program

This script contains the main function, and can be executed from the terminal with the following arguments:

- `source`: the video stream source; can be a file or `0` for the webcam;
- `save` *(optional)*: can be `True` or `False`; it specifies whether the processed video should be saved on disk;
- `destination` *(optional)*: if the processed video is going to be saved, the user can specify the output file name with this argument (otherwise the system will assign an appropriate one);
- `setup_file` *(optional)*: specifies the `INI` file to be used during initialization, if different from the default one (`setup.ini`); useful if the user has many different cameras or setups.

As mentioned in section 2.3.1, this script reads the `INI` file to load all the needed parameters and checks if there are any reference points saved in there, then proceeds as explained in said paragraph. It uses a `mode` flag (`True` for floor corners, `False` for

mat) in order to determine, in each subsequent function, which coordinates should be used (floor or mat), since the transformations needed for the two mechanisms are different.

The script then opens the given video stream by using the OpenCV `VideoCapture()` method[14]; since this function is sometimes prone to error, the script will make a certain number of attempts[4] to open it before returning an error and exiting.

After eventually creating an OpenCV `VideoWriter` object to save the processed video (if the `save` argument was `True`), the program passes the first frame to `process_frame_first()`, thus delegating to this function the map generation, then proceeds to feed the subsequent frames to the similar `process_frame()` function, in a loop.

Once the video has ended, or the user has chosen to exit the program by pressing `ESC`, the script safely exists by saving the video to disk (if needed) and closing the video window before exiting.

3.2 Setup

The `read_ini.py` script unifies the parsing of the `setup.ini` file in a single function, in order to better process the data it contains; it stores these parameters in a series of "themed" dictionaries (`system`, `map`, `mat`, `overlay` and `status bar`) in order to access only once the disk for the settings' retrieval. For this purpose, the file also contains a few utility functions to process some data types found in the `INI`, such as colors and point coordinates.

Even though the setup file provided with the program contains adequate default values for all parameters, a complete explanation of all variables therein present is given within the `read_ini.py` file itself.

3.3 Frame processing

As the name suggests, this script contains the two functions needed for the complete processing of each video frame, including the OpenPose application (it begins with the import of precisely this library - and throws an error if it does not find it).

3.3.1 *process_frame_first()*

The `process_frame_first()` function takes in input many parameters from the main function, of which the most important are the OpenCV `VideoCapture` and `VideoWriter` objects, the `mode` flag and many settings read from the `INI` file.

[4]Specified in the setup file under the `max_attempts` parameter.

After checking the validity of the video frame that has read, it calls the `get_points()` function[5] in order to set the reference points needed for the homography, then the `generate_overlay()` function[6] for the map and status bar generation.

The `process_frame_first()` function proceeds by calculating and setting a few other parameters further needed, then gets the keypoints of the detected people by using the OpenPose library. It then gives these keypoints, along with other data from the dictionaries previously created by the `read_ini()` function, to the `transform_coords()` function[7], in order to obtain the position on the map of each person and if there are any distance violations. The script proceeds with applying the overlay on the frame through the `apply_overlay()` function[6], save it if needed and display it on the screen; it finally returns control to the main script in order for the program to proceed with the rest of the video.

3.3.2 *process_frame()*

This function basically duplicates the code of `process_frame_first()`, except that it only executes the part after (and including) the OpenPose library, since all other data (overlay and map creation) has been previously generated. It is the function that processes every other frame following the first one, and the only reason for its existence is to avoid checking every time if the overlay has yet to be created (first frame) or has already been generated (subsequent frames).

3.4 Points retrieval

The retrieval of the floor or mat corners, as explained in section 2.3.1, makes use of the `get_points()` function found in the homonymous script. This it the method that allows the user to choose the points by clicking on the image if none were found in the setup file.

The mouse handling and keyboard input part have been taken and improved from the `utils` script found at [19]. Specifically, the initial function has been expanded[3] to allow the user to press `SPACEBAR`[13] in order to add borders to the image (if the floor mode has been selected), to limit the number of points to four (it previously had no limits) and to properly check that the user has selected all corners before continuing (thus avoiding a potential program crash, see fig. 11 and 12). The four points are also drawn in four different colors on the image, as opposed to the original version of the script.

[5]Found in the `get_points.py` script.
[6]Found in the `overlay.py` script.
[7]Found in the homonymous script.

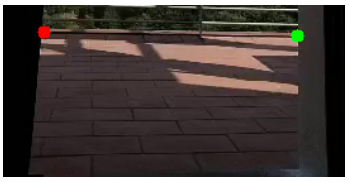

Figure 11: In this window the user has selected only two points so far; an error message will be displayed in the terminal if he or she presses `ENTER` before selecting the remaining two.

```
Click on the four corners of the floor plane (top left, top right, bottom right, bottom left) then press ENTER,
or press SPACEBAR to add or change borders.
Otherwise, press ESC to exit.

Invalid key or not enough points selected (points left: 2). Press ENTER to continue.
```

Figure 12: An example of the error message displayed in the terminal if the user presses `ENTER` before selecting all four corners; the program will not crash, but wait for the remaining mouse clicks.

3.5 Overlay generation and application

The `overlay.py` script contains the `generate_overlay()` and `apply_overlay()` functions previously mentioned in sections 3.3.1 and 3.3.2. Unlike in earlier paragraphs, where the status bar also included the map, in this section a distinction will be made between the words *map*, which is going to designate the actual map generated by the program, and *status bar*, which is going to denote only the grey bar containing the informative text.

3.5.1 *generate_ overlay()*

The `generate_overlay()` function begins by defining the maximum map dimensions, determined from the video resolution; the map will have the same width as the frame width minus the minimum status bar width[8], while its maximum height is calculated according to a specified percentage of the frame height, that can be found in the `INI` file.

The function proceeds with calling what is probably the single most important operation in the program, namely the `warp()`[9] method that executes the homography and returns the transformation matrix and warped image (the latter is going to be resized into the actual map).

[8]The status bar has a minimum width of 200 pixels, needed in order to correctly fit all text.
[9]Defined in the homonymous Python script.

At this point `generate_overlay()` checks if said image is larger than the maximum map dimensions calculated in the beginning, and proceeds to scale it accordingly, then eventually adds a horizontal border if the resulting height is smaller than the status bar height (see fig. 13 and 14).

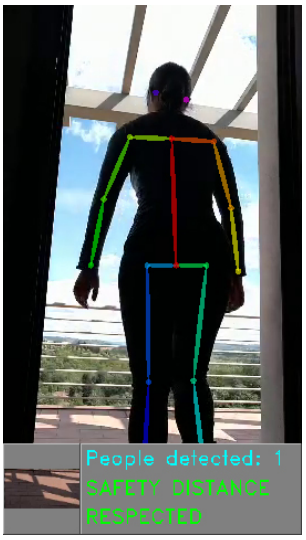

Figure 13: Horizontal map with top and bottom borders.

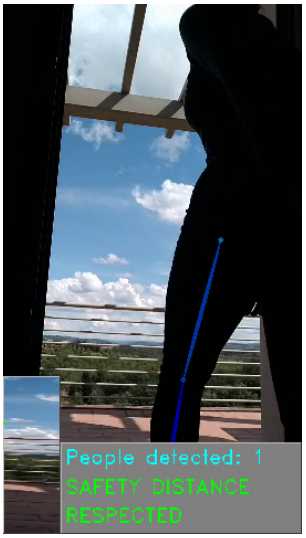

Figure 14: Dummy vertical map (shows how the GUI handles it).

In order to have a flexible overlay that will not impede a correct visualization of the scene after its application, the map and status bar can assume a total of four different positions on the video, as shown in figures 15, 16, 17 and 18.

Figure 15: Status bar with overlay position n°0.

Figure 16: Status bar with overlay position n°1.

Figure 17: Status bar with overlay position n°2.

Figure 18: Status bar with overlay position n°3.

The next operations add a thin border[10] around the map and status bar as a graphical improvement and compute their corners according to the chosen overlay position. This information, together with a few other parameters derived from the `warp()` function (among which the homography matrix and final map image), are added to the `overlay_data` dictionary alongside the existing parameters read from the setup file. These variables will be used for the application of the overlay in each subsequent frame and to ensure the correct processing of the keypoints returned by the OpenPose library since, as seen, the map is not just a simple homography of the first video frame, but undergoes a series of resizings in order to adapt it to the overlay itself.

3.5.2 *apply_ overlay()*

This function, as its name implies, applies an existing map and its status bar to each frame of the video. By using the `overlay_data` dictionary read from the `INI` file and updated after the overlay generation, together with the data about the people's position calculated within the two `process_frame` functions, it draws the map, status bar and informative text over each video frame according to the data it receives.

It is worth noting that this function further adjusts the calculated position of the people returned by `transform_coords()` by calling on this data the `adjust_position ()` function[11], in order to draw the coloured dots consistently with the map position (which, as specified in section 3.5.1, can be in any of the four corners of the frame).

[10] Its thickness can actually be defined in the setup file.
[11] Found in the `transform_coords.py` script.

This method also uses a tolerance[12] to enable the program to draw on the edge some points that fall not too far outside the map, a feature needed in those cases where OpenPose returns keypoints slightly off the real position of the person they refer to (for example when the person lifts a foot, see fig. 19).

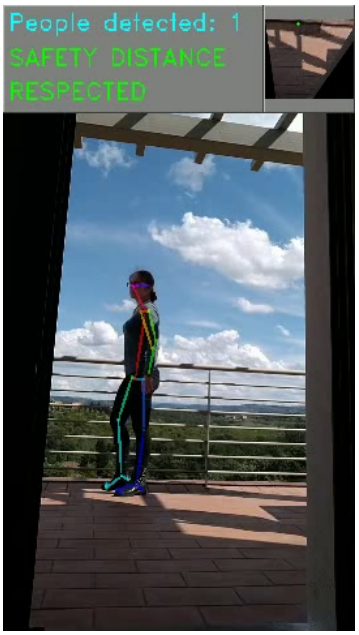

Figure 19: In this particular frame the estimated position of the person would fall slightly outside the top edge of the map (around 5 or 6 pixels from the edge); her position is nonetheless displayed correctly because a tolerance of 20 pixels has been set in the setup file.

3.6 Homography

A homography is a perspective transformation of a plane, that is, a reprojection of a plane from one view into another[18]. It has many practical applications in computer vision, such as image rectification, image registration, or computation of camera motion - rotation and translation - between two images [1].

The homography matrix H is a 3×3 matrix that relates two points on different planes as follows[26]:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1}$$

[12]Specified, in pixels, in the `INI` file.

In other words, given a point with coordinates (x, y) in the source image, H allows to calculate the coordinates (x', y') of the corresponding point in the destination (projected) image[13]; if applied to the whole source image, that is to every source point, the homography matrix returns the destination image in its entirety.

In order to be able to use it, however, one needs to first compute H; the simplest way to do this is to use the coordinates of at least four corresponding points (points that lie on the same plane but are viewed from a different perspective, see fig. 20): $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ and $\{(x'_0, y'_0), (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3)\}$.

The importance of the source and destination points, extensively discussed in sections 2.1.1, 2.3.1 and 3.4, is now clear since the OpenCV library provides a useful function to calculate a homography matrix given these corners, namely `findHomography()`; the matrix returned by this method can be subsequently used to transform any other point in the source image, and as further explained is the basis for this program's map creation and estimation of the people's position.

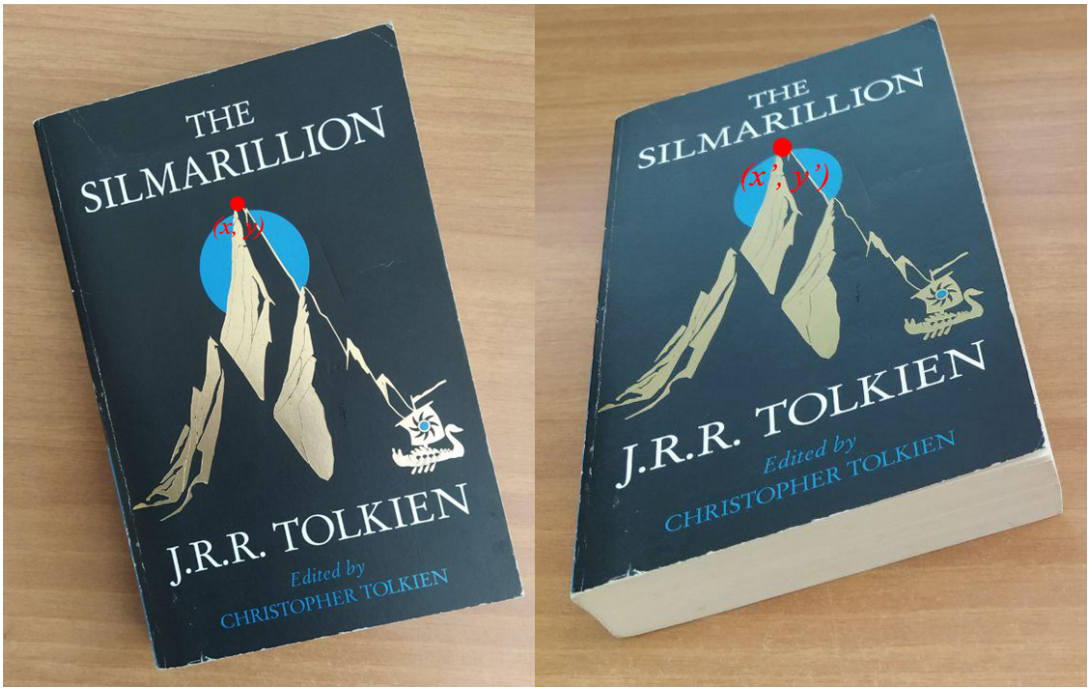

Figure 20: An example of corresponding points in the source image (left) and destination image (right).

[13]The destination image could simply be a different view of the scene represented in the source image, as shown in figure 20.

3.6.1 *warp()*

As mentioned earlier, the `warp()` function executes the actual geometric transformation needed to obtain the map from the original video frame, along with the data further needed by the `generate_overlay()` method in order to create a valid overlay. Its functionality depends on whether the user has chosen to use floor or mat reference points, but can be outlined in both cases with the following steps:

1. [14]Check if destination points are available (see section 2.3.2); if they are, go to step 3;
2. Generate destination points;
3. Get homography matrix using the OpenCV `findHomography()` function;
4. [15]Translate the homography matrix in order to apply it to the whole frame; this step is needed because otherwise the `warpPerspective()` function would crop an essential part of the original frame[16] (see fig. 21 and 22 for an example); t_{02} and t_{12} are set according to the original image width and height:

$$H' = T \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{02} \\ 0 & 1 & t_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \tag{2}$$

5. Apply homography to the original video frame using the OpenCV `warpPerspective()` function;
6. [15]Crop the warped image to contain only the region of interest; this operation is needed because the previous translation causes the homography to be applied to the whole frame, of which only the mat and a given region of interest around it is actually needed for the map.

Figure 21: Correct map of the requested region of interest generated from mat corners after having translated the homography matrix.

Figure 22: Cropped map generated from the same mat corners by the `warpPerspective` function if the homography matrix is not translated.

[14]Floor case only.
[15]Mat case only.

For debug purposes, the warped image (which is different from the final map because, as previously seen, more resizing occurs in the `generate_overlay()` function) can be shown if a parameter `show = True` is passed to `warp()`. However, sometimes this image would be too big to fit the screen, so the program automatically resizes it according to the display resolution that is detected through the `screeninfo` library[28].

The script finally returns the warped image and the homography matrix (the translated version in the mat case), along with the parameters needed for the subsequent transformations (namely, in the mat case, an offset introduced by the translation and the map dimensions in centimeters, which have been automatically calculated from the mat's own dimensions).

3.7 People's position estimation

Both `process_frame` functions use an external script to obtain and process the people's position, specifically the `transform_coords.py` script. This program takes in input the OpenPose keypoints along with the homography matrix and other variables calculated by the `generate_overlay()` function (that are needed to ensure consistency with the subsequent transformations that the original warped image undergoes in the same function), and returns an array containing the pixel coordinates of each person detected in the frame along with the distance (in centimeters) between each of them.

The program is quite simple: after getting the average position of the two heel keypoints of each person[16], it applies the homography matrix to them by using the OpenCV `perspectiveTransform()`[27] function along with any offsets generated by the homography translation (only in the mat case). The particular choice of the heels was made in order to exploit the fact that OpenPose can detect even people that might partially fall outside the frame, and can prove effective wherever the floor covers the (almost) entirety of the frame and only the people's feet are visible in the furthest area (fig. 23).

An array containing the previous positions of the same people is used to smooth the transition between one frame and the next by applying the following formula, where f is the number of the current frame and α is a parameter that can be set in the `INI` file as the `position_alpha` parameter[17]:

$$x(f) = \alpha \cdot x(f) + (1 - \alpha) \cdot x(f - 1) \tag{3}$$

$$y(f) = \alpha \cdot y(f) + (1 - \alpha) \cdot y(f - 1) \tag{4}$$

[16] OpenPose keypoints number 21 and 24 for the BODY_25 model[6]

[17] It has been observed that $\alpha = 0,90 \sim 0,95$ achieves the desired effect.

Figure 23: The person shown in this video is detected by OpenPose even if only the lower half of her body is visible.

The measure was introduced in order to reduce the shaky movement of the position drawn on the map caused by the Openpose library which, as one would expect, does not return sufficiently accurate keypoints for each frame, even if the person moves by only a few pixels.

This feature is however limited by the fact that the program does not distinguish between different people, so therefore it only works if OpenPose detects the same number of people between frames and returns them in the same order. In fact, it has been noted that OpenPose *does not* return detected people always in the same order, and this caused noticeable errors in the map position drawing. This complication has been solved in the following naive, yet in practice rather effective way: before applying equations 3 and 4, the array containing the people's positions is reordered by sorting the x coordinates (each of them together with their relative y coordinate) in ascending order. Since a person's position does not usually change too much from a single frame to another, this approach ensures that every position stored in the sorted array corresponds to the same person, every time.

The `transform_coords.py` file also contains a few auxiliary functions, namely `adjust_position()` (used in `apply_overlay()`, see section 3.5.2) and `get_distance()`, needed to calculate the distance between each pair of people detected in each frame and warn about any violations.

3.8 Practical measures for a real-world application

As stated in the abstract of this paper, there have been taken multiple measures in order to create a program less prone to error and ensure an overall smooth user

experience.

Even if these technicalities are best observed directly in the code, a few of them are hereby mentioned as a sample of this effort to limit various errors:

- all parts of the program where user input is required are heavily foolproofed against invalid input, mouse corner selection included (as mentioned in 3.4);
- in the `physical_distance_detector.py` script, the destination name that the user can enter as a command line argument has its extension analysed and corrected if absent or wrong (in the current configuration the OpenCV library can only save videos in the `AVI` format);
- various readable errors are printed in the terminal window if some of the parameters found in the setup file are out of their range, such as the overlay position and the floor/mat source points;
- as seen in the `warp()` function (section 3.6.1), a whole block of code is devoted to checking that the warped image is not larger than the current screen resolution (on either Windows or Linux), and if so, to resizing said image accordingly; this has been implemented as a measure to avoid enormous windows on small screens (such as those that are usually found in surveillance systems, a possible real-world application of this software).

4 System requirements

The system requirements for this software are mainly derived from the demanding requirements of the OpenPose library which, as stated in the official documentation for the NVIDIA GPU version, are the following[10][9]:

- CMake GUI[17];
- NVIDIA graphics card with at least 1.6 GB memory available;
- CUDA 10 (for Ubuntu 18)[22]
- at least 2.5 GB of free RAM memory for BODY_25 model (assuming cuDNN installed);
- highly recommended: cuDNN 7.5 (for Ubuntu 18)[18];
- highly recommended: a CPU with at least 8 cores.

The Caffe dependency is automatically satisfied when downloading the OpenPose source code (which contains it), while all other libraries (OpenCV, numpy[25], screeninfo) can be easily installed using the `pip3` command on both Linux and Windows.

[18]It has been observed, on this project's development system (see 4.2), that the OpenPose library quickly runs out of memory without cuDNN installed, so in this case this requirement one turned out to be actually *highly necessary*.

4.1 OpenPose installation

A whole new paper could be written about the difficulties encountered while installing the necessary requirements for the OpenPose library, but since this is not the purpose of this article, only a brief summary is hereby given in order to inform the user about the possible complications that might arise while trying to compile OpenPose from its source code[8][30].

Windows 10 + AMD graphics

An early intention for this project was to develop it on Windows 10 using an AMD Radeon Vega 64 GPU, which was among the few AMD GPUs supported by OpenPose[11]; however, after intensive trials, this turned out to be infeasible as the OpenPose library kept throwing CUDA errors in spite of the system having no NVIDIA graphics card mounted on board. It has been speculated that this problem arised from the fact that the machine did use, at a certain point in its life, a NVIDIA GPU, and although that had been long uninstalled together with its drivers[19], some traces of the previous installation have probably remained somewhere in the Windows environment and caused OpenPose to believe that it was running on a CUDA system – which it was not. In spite of all the efforts, it has not been possible to completely eradicate these remains without resorting to highly invasive methods (such as reinstalling Windows), which were not an option.

Windows 10 + NVIDIA graphics

The subsequent approach hoped to solve the detection of nonexistent NVIDIA drivers by reinstalling them again and actually using a NVIDIA GPU. Many official[23] and unofficial[29] guides have been followed[20] in order to properly reinstall these drivers but, notwithstanding the efforts, CUDA kept throwing the same error as before. Given that much time had already been invested in these attempts, the author thus deemed counterproductive (see fig. 24) to keep trying without first reinstalling Windows 10, and since this was not possible, a new boot partition has been created in order to move the project's development on Linux.

[19]The removal of all NVIDIA drivers, CUDA and cuDNN software has been accomplished using the official NVIDIA guides and uninstallers which, as stated in this paragraph, seem to actually be insufficient to restore the machine to its previous state. Ulterior research and troubleshooting, even outside of the official NVIDIA channels, did not help in this matter.

[20]The author has read so much material on the subject that it is unthinkable to account for all of it in this paper, even if only for the fact that after a certain point it was impossible to keep trace of all procedures that had been tried; none of it, in any case, solved the problem.

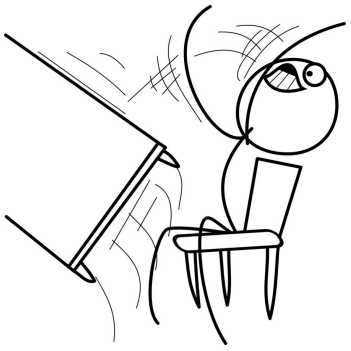

Figure 24: This hilarious meme[20] pretty much sums up the author's reaction after more than a week of attempts to get OpenPose to work on Windows 10. Funnily enough, Ubuntu proved not much better - at least initially.

Ubuntu 18.04 + NVIDIA graphics

Using a clean Linux installation was supposed to facilitate the process of installing the requirements for OpenPose; it turned out that even on Linux, NVIDIA software was still very challenging to install.

The main problem was to be able to get a compatible GPU driver to work with CUDA and vice versa, because they kept conflicting during their own installations even if, according to official documentation[24], they were supposed to get along just fine. cuDNN also posed a problem, and extensive research was needed in order to get CUDA to find it; for the purpose of installing this library, one should be able to confidently read and modify CMake files[4].

After much trial and error, troubleshooting and scouring of the most remote forum posts, a working configuration has been obtained after compiling the OpenPose source code[5][7]; however, given the tremendous amount of attempts and software installed[21], writing a complete guide about this process would be quite problematic and possibly useless, since there are too many dependencies among all the packages that had to be installed and they are continuously updated, thus generating new incompatibilities.

4.2 System specifications

This project has been developed and tested on a system with the following specifications:

- **Motherboard**: MSI MS-B106

[21]It has been estimated that about 15 GB of data have been downloaded in order to install, uninstall and reinstall the NVIDIA software with all its requirements - more than once - until everything worked as was supposed to.

- **CPU**: Intel Core i7-6700K @ 4.01
- **RAM**: 16 GB DDR4 @ 3000 MHz
- **GPU (AMD)**: AMD Radeon Vega 64 *(later replaced with the NVIDIA GPU)*
- **GPU (NVIDIA)**: Zotac GeForce GTX 970 *(courtesy of the University of Florence)*
- **SSD**: Samsung SSD 850 EVO [500 GB]
- **HDD**: WD Blue [3 TB]
- **OS**: Ubuntu 18.04.4 LTS (Bionic Beaver)
- **NVIDIA driver**: v. 440.100
- **CUDA**: v. 10.2
- **cuDNN**: v. 7.6.5

5 Conclusions

The software here presented is able to determine the physical distance between people with satisfying accuracy (around $\pm 5 - 15cm$), and in contrast to other similar programs[15] it also draws a map of the environment where it marks the position of every person. It is uniquely based entirely on geometrical transformations and relies on OpenPose, a solid pose estimation library. Even though the latter represented a great challenge to configure, it resulted to be a complete and easy to implement tool once it was properly working.

5.1 Future developments

The project fully achieves its purpose while introducing new mechanics along the way. In spite of the effort put in making it as complete as possible, however, there is certainly room for improvement; many exceptions, for example, have not been covered (especially some niche errors thrown by the OpenPose and OpenCV libraries), and although the code has been fully refactored before the final commit, it could probably be further optimized for a faster execution.

OpenPose's keypoints could be further elaborated in order to get better positions. However, as the author has observed while working on the position smoothing part (discussed in 3.7), the last improvement would require a lot of comparisons between frames and much additional data should be saved for each frame, and together these operations would greatly reduce overall performance because of the number of calculations that such processing would require for every single video frame.

In the hopes that the final user will not come across these untested exceptions nor will try to run the program on low-end systems, the source code remains available at the project's original repository[21] for someone else to read and eventually work on.

References

[1] Homography (computer vision) - Wikipedia. https://en.wikipedia.org/wiki/Homography_(computer_vision).

[2] OpenCV. https://opencv.org/, 1999.

[3] Aleksandr Rybnikov Alexey Spizhevoy. *OpenCV 3 Computer Vision with Python Cookbook*.

[4] cnzJohn. Cmake openpose problem (cuDNN not found). https://github.com/CMU-Perceptual-Computing-Lab/openpose/issues/486, 2018.

[5] Nisha Gandhi. Real-time Pose Estimation in webcam using OpenPose: Python 2/3 OpenCV. https://medium.com/pixel-wise/real-time-pose-estimation-in-webcam-using-openpose-python-2-3-opencv-91af0372c3 2018.

[6] Gines Hidalgo. OpenPose - Keypoint Ordering in C++/Python. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/output.md#keypoint-ordering-in-c-python, 2020.

[7] Gines Hidalgo. OpenPose - Quick Start. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/quick_start.md#quick-start, 2020.

[8] Gines Hidalgo. OpenPose Installation. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/installation.md#installation, 2020.

[9] Gines Hidalgo. OpenPose Installation - Requirements and Dependencies. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/installation.md#requirements-and-dependencies/, 2020.

[10] Gines Hidalgo. OpenPose Installation - Ubuntu Prerequisites. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/prerequisites.md#ubuntu-prerequisites, 2020.

[11] Gines Hidalgo. OpenPose Installation - Windows Prerequisites. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/prerequisites.md#windows-prerequisites, 2020.

[12] Gines Hidalgo. OpenPose Python Module and Demo. https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/modules/python_module.md, 2020.

[13] Abid Rahman K. Using other keys for the waitKey() function of opencv. https://stackoverflow.com/questions/14494101/using-other-keys-for-the-waitkey-function-of-opencv, 2013.

[14] Alexander Mordvintsev Abid K. Getting Started with Videos - OpenCV. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_video_display/py_video_display.html, 2013.

[15] Gurucharan M. K. COVID-19: AI-Enabled Social Distancing Detector using OpenCV. https://towardsdatascience.com/covid-19-ai-enabled-social-distancing-detector-using-opencv-ea2abd827d34, 2020.

[16] kbarni. OpenCV - Perspective transform without crop. https://answers.opencv.org/question/144252/perspective-transform-without-crop/, 2017.

[17] Kitware. CMake GUI. https://cmake.org/.

[18] Frank Lamosa. What is a homography, and how is it calculated? - Quora. https://www.quora.com/What-is-a-homography-and-how-is-it-calculated, 2017.

[19] Satya Mallick. Homography Examples using OpenCV (Python/C++). https://www.learnopencv.com/homography-examples-using-opencv-python-c/, 2016.

[20] Know Your Meme. Flipping Tables. https://knowyourmeme.com/memes/flipping-tables, 2011.

[21] Paula Mihalcea. Physical Distancing Detector. https://github.com/PaulaMihalcea/Physical-distancing-detector, 2020.

[22] NVIDIA. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[23] NVIDIA. CUDA Installation Guide for Microsoft Windows. https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html, 2020.

[24] NVIDIA. NVIDIA CUDA Installation Guide for Linux. https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html, 2020.

[25] Travis Oliphant. Numpy. https://numpy.org/, 2005.

[26] OpenCV. OpenCV - Basic concepts of the homography explained with codem. https://docs.opencv.org/3.4/d9/dab/tutorial_homography.html.

[27] OpenCV. OpenCV - Operations on Arrays: perspectiveTransform. https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#perspectivetransform, 2011.

[28] rr. screeninfo 0.6.5. https://pypi.org/project/screeninfo/, 2020.

[29] Kavinda Senarathne. CUDA Toolkit on Windows 10. https://medium.com/analytics-vidhya/cuda-toolkit-on-windows-10-20244437e036, 2019.

[30] Erica Zheng. A 2020 Guide for Installing OpenPose. https://medium.com/@erica.z.zheng/installing-openpose-on-ubuntu-18-04-cuda-10-ebb371cf3442, 2019.

Figure 25: Despite repeatedly violating the safety distance (of course, for research purposes), these test subjects (Marco & Riccardo Pratelli) had their share of fun while recording the sample videos. The author kindly thanks them for their contribution.