Auto-Censoring for Video Calls

HCI final project

Sarath Kondeti
(115223356)

Abstract

Due to the pandemic, a lot of people have started working from home. And privacy has become one major concern. Unlike office environments, homes and other public spaces are occupied by other people who would not prefer to be captured during video calls unintentionally. There are many reasons they would like to keep their identity intact. So, we designed a system that automatically detects & censors unknown people walking into the camera frame during an ongoing video call. The result was a smooth background process that does not interrupt the subject or the members in the video call and runs without significant CPU load or FPS loss in the video output. Now, anyone can walk near a person having a video call without having to worry about their identity being revealed.

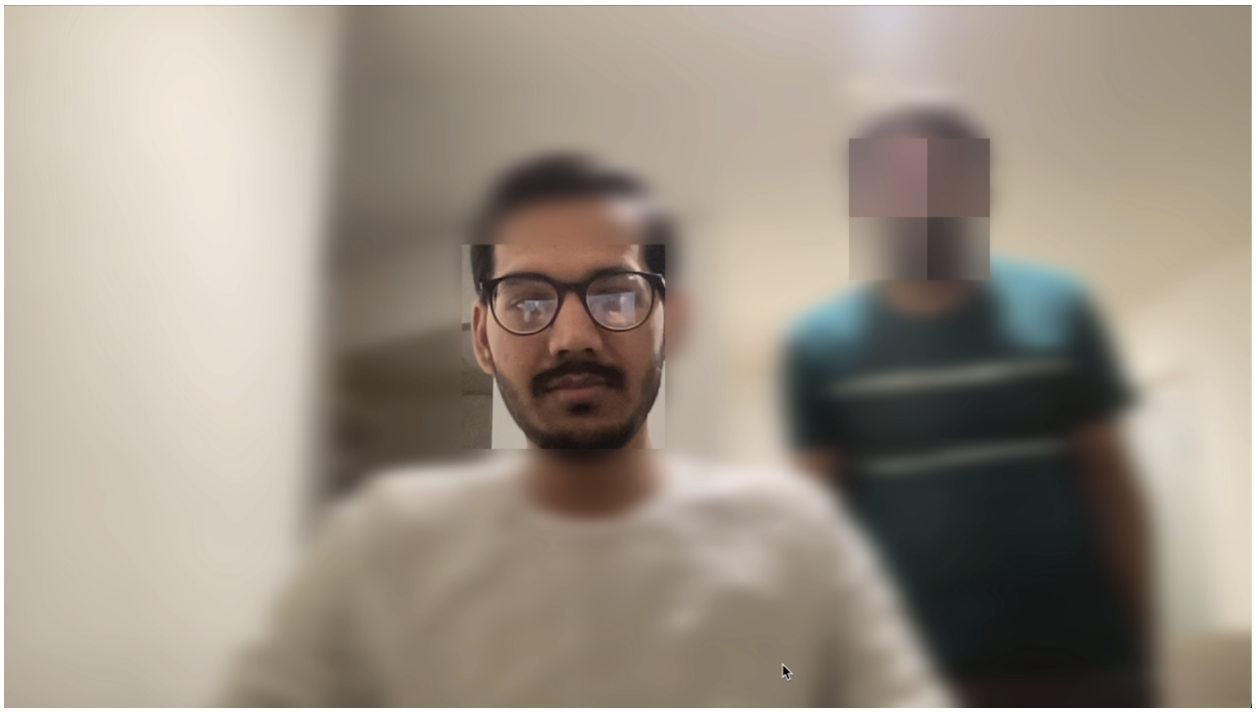


Figure shows our system censoring the unknown person in the frame while keeping the subject intact.

Introduction

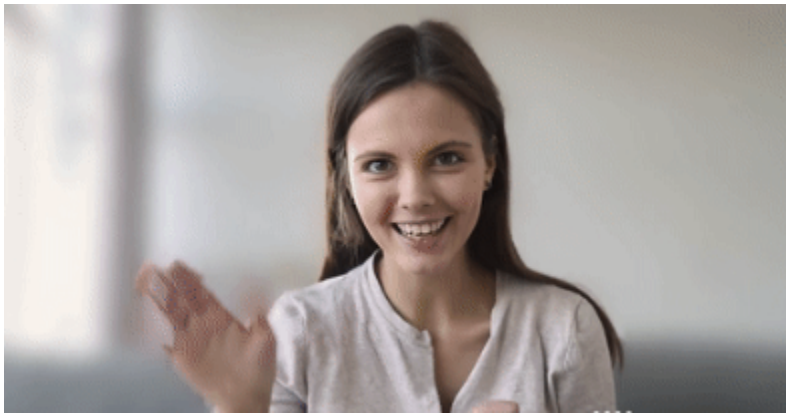
A decade ago people used to work at their offices and have their meetings and conferences physically but due to the pandemic, people were forced to work out of their homes which demanded them to come up with new ways of interacting with their coworkers. Most of them could get away with a few voice calls but many fields required a visual exchange of information so we started using video conferences using Google meets, Zoom, etc. This solved most of the troubles but one was remaining. People preferred looking at people while speaking and so everyone on the call had to enable their webcam to let others see them. Enabling webcams gives rise to a whole array of problems. One major problem is the security of people or things in the environment.

If the personnel is having a video call with some third party, the webcam can capture some confidential information present in the background or it can capture someone who prefers to stay anonymous. Security issues also exist when video call happens in a home environment. These may not be confidential but they might still be private and intimate. It can either be personal things or people. Here we address how we protect the identity of other people during online conferences. We often refer to the “work from home” scenario but obviously, this can be applied to other situations.

During work from home, other habitats had to be aware when roaming around a person having a video call for various reasons like preserving their identity, not interrupting the members of the video call, etc. Until now the user had to be aware of the surroundings and turn off the video when someone walks in or request them not to approach. This often interrupts the user from paying attention to the conference. Our system takes care of this automatically thus helping the user to focus on the call instead of being distracted. The system detects movement and automatically blurs the background until they leave. It also pixelates their face as an extra measure. All this happens while the user’s image is kept intact.

Related work

There already exist many solutions for this problem but they often have some side effects. One of them is to have the user enable the fake background or background blur. Fake background is somewhat better compared to a background blur in terms of security but it also makes the image too artificial. They also have one major downside which is their implementation. The way these tools work is that they separate foreground objects from background objects to get the outline of the user while blurring the background. This fails in the security aspect when someone who wants to stay anonymous walks too close to the camera without realizing it. It seems these tools are focused to cover the clutter people often have in their environment from the members in the video conference and are not meant for preserving privacy. Another downside is that they require the user to actively enable and disable when need be.



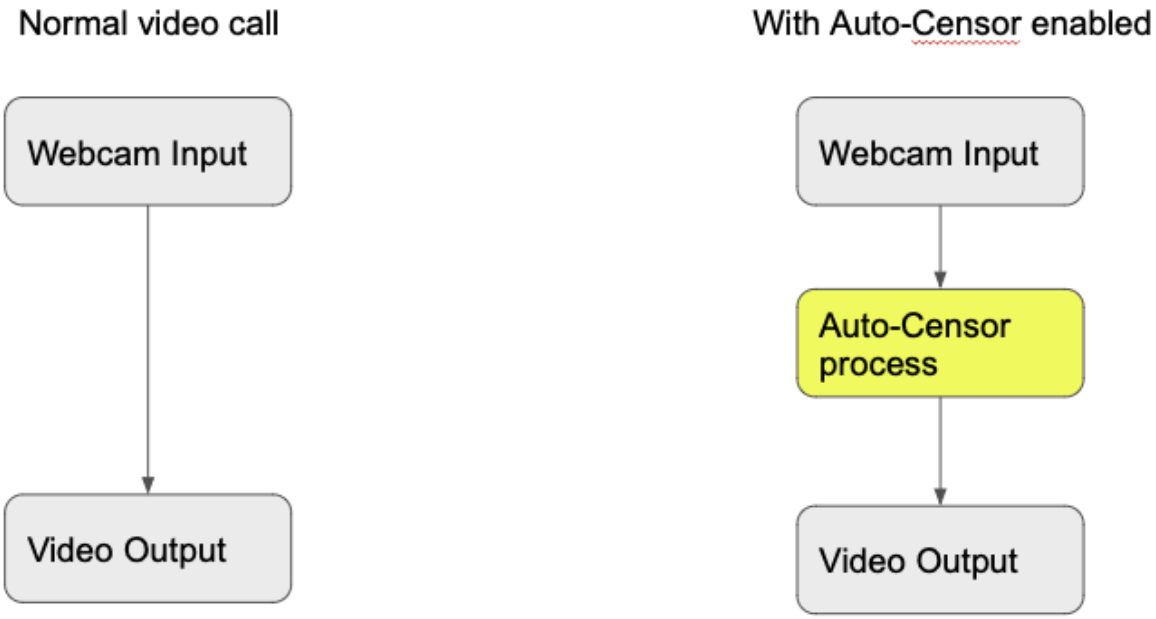
Background blur

Fake background

Implementation

Architecture
Just like fake background and background blur tools, we need to capture raw frames from the webcam and process them before outputting the video. And this processing as a toll on the CPU load as well as the FPS output. But we discuss some optimizations in later sections which bring the loss to only ~10-15 frames per second.

Below we see a basic architecture of our process compared to the status quo.



We used the open-cv python library for capturing webcam data.
video_capture = cv2.VideoCapture(0)
ret, frame = video_capture.read()

We process the raw frame and output as below.
cv2.imshow('Video', frame)

Facial Recognition

This is the most important aspect of the project since it decides which faces to censor and which not to. We use the face_recognition library which was built on the dlib library. Our idea is to load the user's facial image beforehand. So, if it finds any other unrecognized face it can censor them.

keshav_image =
face_recognition.load_image_file("keshav.jpg")
keshav_face_encoding =
face_recognition.face_encodings(sarath_image)[0]

known_face_encodings = [
keshav_face_encoding
]

All the loaded faces are encoded and maintained in a list.

Next, we find all the faces in a captured frame.
Then we encode them and find the best matches in **known_face_encodings.**
If there are no best matches, we name them "unknown" so, we can separate recognized and unrecognized faces.

```
# Resize frame of video to 1/4 size for faster face recognition processing
small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)
# Convert the image from BGR color (which OpenCV uses) to RGB color (which face_recognition uses)
rgb_small_frame = small_frame[:, :, ::-1]
# Find all the faces and face encodings in the current frame of video
face_locations = face_recognition.face_locations(rgb_small_frame)
face_encodings = face_recognition.face_encodings(rgb_small_frame, face_locations)
face_names = []
for face_encoding in face_encodings:
    # See if the face is a match for the known face(s)
    matches = face_recognition.compare_faces(known_face_encodings, face_encoding)
    name = unknown # default name
    # # If a match was found in known_face_encodings, just use the first one.
    # if True in matches:
    #     first_match_index = matches.index(True)
    #     name = known_face_names[first_match_index]
    # Or instead, use the known face with the smallest distance to the new face
    face_distances = face_recognition.face_distance(known_face_encodings, face_encoding)
    best_match_index = np.argmin(face_distances)
    if matches[best_match_index]:
        name = known_face_names[best_match_index]
    face_names.append(name)
    if name is unknown:
        last_unknown_detected = time.time()
    else:
        last_authorized_detected=time.time()
```

There is still one problem, the face_recognition has some limitations. It can not recognize faces if they are far behind in the background. So, we can not censor if we can not recognize them. In this scenario, we use another approach. The open-cv library has a nice utility that can we can make use of. “**cv2.CascadeClassifier**” can find any prespecified object in the image such as eyes, faces, bodies & upper bodies. We preload the classifier with an upper-body cascade.

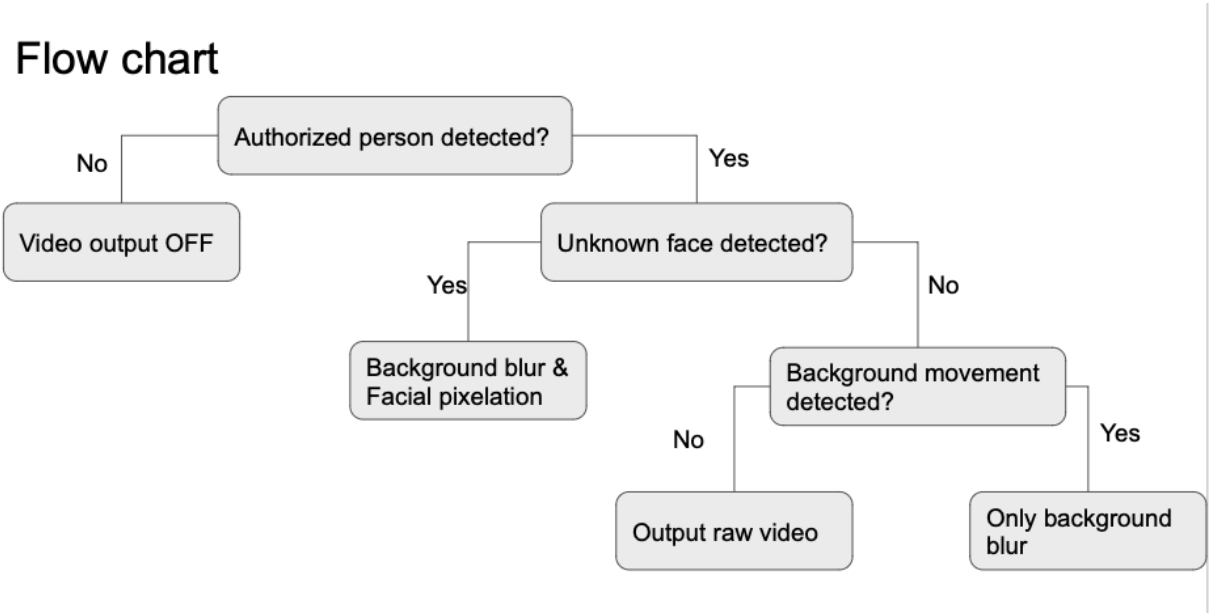
bodyCascade = cv2.CascadeClassifier("haarcascade_upperbody.xml")

```
def backgroundMovement(frame):
    global last_unknown_detected
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    bodies = bodyCascade.detectMultiScale(
        gray,
        scaleFactor = 1.4,
        minNeighbors = 5,
        minSize = (50, 100), # Min size for valid d
        flags = cv2.CASCADE_SCALE_IMAGE
    )
    if len(bodies)!=0:
        last_unknown_detected    = time.time()
        return True
    return False
```

We adjust the parameters such that it does not recognize the user’s upper body because they are sitting closer.

Finally, with the above two tools, we can find out if they are unknown people in the frame. Either their face will be recognized or their upper body will be recognized if they are in the background.

Next, we will discuss how we censor unknown people now that we can recognize their faces in the frame.



Here, we discuss our censoring criteria.

- If the user has stepped away from the computer, we disable the output. This is a safe option because even if an unknown person walks by they won't be captured. No censoring is required in this case.

- If the user is present in front of the camera, we check whether any other unknown face is detected. If yes, we blur the background as well as pixelate their face as an extra measure.

- If no unknown faces are detected but there's movement in the background. We only blur the background.

Once we have face positions in the frame, we use Numpy slicing to extract the face from the frame.

- Case “User’s face” - we save it without any alteration.
- Case “unknown face” - we apply pixelation to it.

The original frame will be blurred using the blur_img method. Now we paste the faces(altered & unaltered) back on the frame at the same position they were taken from.

Below we see that blurred_frame is the result after “frame” goes through blurring. And detected_face goes through pixelation if its name is “unknown”. After that, they are pasted back on blurred_frame.

```
blurred_frame = blur_img(frame, factor = 15)
for (top, right, bottom, left), name in zip(face_locations, face_names):
    # Scale back up face locations since the frame we detected in was scaled to 1/4 size
    top    *= 4
    right  *= 4
    bottom *= 4
    left   *= 4
    detected_face = frame[int(top):int(bottom), int(left):int(right)]

    if name == unknown:
        pixelated_face = detected_face.copy()
        width = pixelated_face.shape[0]
        height = pixelated_face.shape[1]
        step_size = 80
        for wi in extract_indexes(width, step_size):
            for hi in extract_indexes(height, step_size):
                detected_face_area = detected_face[wi[0]:wi[1], hi[0]:hi[1]]

                if detected_face_area.shape[0] > 0 and detected_face_area.shape[1] > 0:
                    detected_face_area = blur_img(detected_face_area, factor = 0.5)
                    pixelated_face[wi[0]:wi[1], hi[0]:hi[1]] = detected_face_area

        blurred_frame[top:bottom, left:right] = pixelated_face
    else:
        blurred_frame[top:bottom, left:right] = detected_face
        # Draw a box around the face
```

Image Blur

We used the cv2 library once again for GuassianBlur() method. We can use the factor parameter for adjusting the intensity of the blur.

```
# this function outputs blurred img of input, use factor to agjust blur intensity.
def blur_img(img, factor = 20):
    kW = int(img.shape[1] / factor)
    kH = int(img.shape[0] / factor)
    #ensure the shape of the kernel is odd
    if kW % 2 == 0: kW = kW - 1
    if kH % 2 == 0: kH = kH - 1
    blurred_img = cv2.GaussianBlur(img, (kW, kH), 0)
    return blurred_img

def isPresent(s,arr):
    for x in arr:
        if x==s:
            return True
    return False
```

Image Pixelation

Pixelation is done in a similar way as blurring. We divide the image into small blocks and individually apply gaussian blur to them and put them back in the same location.

```
pixelated_face = detected_face.copy()
width = pixelated_face.shape[0]
height = pixelated_face.shape[1]
step_size = 80
for wi in extract_indexes(width, step_size):
    for hi in extract_indexes(height, step_size):
        detected_face_area = detected_face[wi[0]:wi[1], hi[0]:hi[1]]

        if detected_face_area.shape[0] > 0 and detected_face_area.shape[1] > 0:
            detected_face_area = blur_img(detected_face_area, factor = 0.5)
            pixelated_face[wi[0]:wi[1], hi[0]:hi[1]] = detected_face_area

blurred_frame[top:bottom, left:right] = pixelated_face
```

Finishing touches

Our laptop's webcam is not the best hardware for facial recognition. Similarly, the python libraries are not optimized for our use case. So, facial recognition and body detection would not work on some intermittent frames. This may lead to flickering of blurring/pixelation effects. So, we introduced some delay variables throughout the code. Such as **last_unknown_detected** - used for keeping the blurring effect on for 3 seconds after the last time movement was detected in the background. **last_authorized_detected -** for keeping the video on for 3 seconds after the last time the user was detected.

Optimizations

We only take alternate frames for facial recognition to reduce the load on the CPU. This does not have any side effects as we still do facial recognition around ~20 times per second.
We scale down the image 4 times so, facial recognition can work a lot faster. But we still output the frame at the original resolution.

Validations

We address a very core security/privacy issue during everyday video calls done by millions of people. Our implementation does not need extra hardware because it makes use of existing data already available through the camera. This implementation can be easily tweaked to adapt to a different use case. Suppose a person wants to stay anonymous during a video call/conference/interview. We easily pixelate their faces in real-time. With this system running, the user does not have to bother turning their camera on/off when they are stepping away.

Future work

We can extend this idea of censorship further to the voice aspect. There might be some confidential conversation ongoing in the background which would not be safe if it were recorded during a video call. Similarly, in a home environment, there might be private/intimate conversations in the surroundings which would be ideal for others in the online call to listen. So, we come up with a similar approach. We preload the user's voice profile and use voice diarization to distinguish between the user's voice and an unknown person's voice. The brief idea is the mute the microphone when an unknown voice profile is picked up by the microphone. Unfortunately, real-time voice diarization is still in the research phase and can't be implemented efficiently yet.

Conclusion

In conclusion, what we made is a robust security-based video calling API. We can easily change our implementation to an API because the input & output for the program are very clear as they are camera input and video output. The tool is quite concise and effective when it comes to preserving privacy among other use cases. People often don't give enough attention to protecting privacy. Our system helps achieve this automatically in the background with minimal setup. Now, the user as well as other people can rest assured that their identity is secure and is not unintentionally being broadcasted somewhere.

References

https://pylessons.com/remove-background
https://sefiks.com/2020/11/07/face-and-background-blurring-with-opencv-in-python/
https://pypi.org/project/face-recognition/