# Milestone 3.3: Webcam Capture Integration (PC Recording)

## Overview and Goals

In this milestone, we add **webcam capture and recording** capability to the PC application. The PC will use its own webcam as an additional video source, recording a video stream in sync with the smartphone cameras. This provides an extra perspective (e.g. a front-facing view of the subject) to complement the phone recordings. The goals of Milestone 3.3 are:

* **Webcam Access:** Utilize the PC’s built-in or USB webcam through OpenCV (or PyQt’s multimedia module) to capture frames.
* **Live Preview:** Display the webcam feed in the PC app’s GUI so the user can monitor it (similar to how phone camera previews might be shown).
* **Recording Synchronization:** When a recording session starts, the PC’s webcam feed is recorded to a video file, saved alongside the phone videos. The PC’s recording should start/stop in sync with the phones.
* **Session Management:** Introduce a “session” concept on the PC – each recording session has a unique identifier (name or timestamp), a dedicated folder for all files (videos, logs, etc.), and coordinated start/stop across devices.
* **Testing & Verification:** Ensure the webcam preview works and that recorded video files are saved correctly and are time-synchronized with the other devices.

By completing this milestone, the PC application will be able to **record from its own camera** in addition to commanding the phones, thereby expanding the multi-camera recording system’s capabilities.

## Development Setup and Prerequisites

Before implementing the webcam integration, make sure your development environment is prepared:

* **Programming Environment:** We assume you are using Python 3.x with PyQt5 (for the GUI) and OpenCV (opencv-python library) for video capture. Ensure these packages are installed. For example, install via pip:
* pip install PyQt5 opencv-python
* (If you haven’t already installed OpenCV and PyQt5 in your project environment, do so now.)
* **IDE/Project Configuration:** Use an IDE like PyCharm or VSCode for easier project management. In your IDE, set the project interpreter to the environment that has PyQt5 and OpenCV installed. If using PyCharm, you can add these packages in **Preferences > Project Interpreter**. Also, ensure that the PYQT5 and cv2 modules import without errors in a Python console.
* **Project Structure:** Organize your code with modules for clarity. For example, you might have a project layout like:
* project\_root/  
   ├── main.py # Application entry point  
   ├── gui\_mainwindow.py # GUI main window logic (PyQt)  
   ├── webcam\_capture.py # Module for webcam thread & recording  
   ├── session\_manager.py # Module for session handling (folders, naming)  
   └── devices\_controller.py# (optional) manages connected phone devices
* You can adjust according to your existing codebase. The key is to plan adding a **WebcamCapture** class (for capturing & recording webcam video) and updating the GUI to include a preview area for the webcam.
* **Hardware:** Obviously, ensure the PC has a webcam connected. Most laptops have an integrated webcam accessible as device index 0. If you have an external USB camera, it might register as index 1, etc. We will default to the first camera (index 0) and allow changing if needed.

With the environment ready, we can proceed step-by-step to implement the webcam capture feature.

## Step 1: Accessing the PC Webcam with OpenCV

**Goal:** Initialize the PC’s webcam and verify we can capture frames from it.

1. **OpenCV VideoCapture:** We will use OpenCV’s cv2.VideoCapture API to access the webcam. In a suitable place in your code (e.g., in the new webcam\_capture.py module or directly in the thread class we’ll create), start by creating a capture object:

* import cv2  
  cap = cv2.VideoCapture(0)
* Here 0 is the device index for the default webcam. If you have multiple cameras or if 0 doesn’t work, try 1 or other indices. You can test this in isolation by running a short script to grab a frame and display it using cv2.imshow (during development) or print the frame size to confirm the camera opens.

1. **Verify Camera Access:** Always check if the camera opened successfully:

* if not cap.isOpened():  
   print("Error: Could not open webcam")  
   # handle the error (e.g., exit or notify user)  
  else:  
   print("Webcam opened successfully")
* If this fails, ensure no other application is using the webcam and that the correct device index is used.

1. **Camera Properties (Optional):** You can retrieve or set properties like resolution or frame rate if needed. For example:

* width = cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH)  
  height = cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT)  
  fps = cap.get(cv2.CAP\_PROP\_FPS)  
  print(f"Default resolution: {width}x{height}, FPS: {fps}")
* You might limit the resolution for performance. For instance, if the default is 1080p but you only need 720p for a preview, you can do cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 1280) and similarly for height (720). Adjusting resolution can reduce CPU usage.

1. **Choose Capture Method (OpenCV vs Qt):** We opt for OpenCV because it gives flexibility (and we likely already use it for calibration or image processing). PyQt5 does have QCamera and related classes in QtMultimedia that can capture video as well. However, using OpenCV allows us to easily process frames (e.g., for potential future enhancements) and use a single code path for both preview and recording. The drawback is we must manually handle threading and conversion to Qt images, which we will do in upcoming steps. (If PyQt’s QCamera were used, it integrates with Qt’s signal/slot but can be more complex to ensure cross-platform codecs for recording. For now, OpenCV is a simpler route.)

At this point, we have the basic handle to the webcam. Next, we will set up a dedicated thread to continuously capture frames from this VideoCapture without freezing the GUI.

## Step 2: Implementing a Webcam Capture Thread for Live Preview

**Goal:** Run the webcam capture loop in a separate thread so that it doesn’t block the GUI, and prepare the frames for display. We will create a new class (e.g., WebcamCaptureThread) that extends QThread and handles reading frames continuously.

Why a thread? Continuously reading the webcam is a blocking operation that runs in a loop. If done on the main thread, it would freeze the GUI. Using a QThread for camera capture allows the GUI to remain responsive[[1]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=Since%20reading%20the%20camera%20feedback,the%20GUI%20of%20our%20application). We emit frames from the thread and update the UI in the main thread.

**Implementation steps for the thread:**

1. **Subclass QThread:** In your webcam\_capture.py (or equivalent), define a class WebcamCaptureThread(QtCore.QThread). Import QThread and pyqtSignal from PyQt5.QtCore. For example:

* from PyQt5.QtCore import QThread, pyqtSignal  
  import cv2, time  
  class WebcamCaptureThread(QThread):  
   frame\_signal = pyqtSignal(object) # will emit frames as QImage or pixmap, or as raw frame if you prefer  
   def \_\_init\_\_(self, parent=None):  
   super().\_\_init\_\_(parent)  
   self.cap = None  
   self.is\_running = False  
   self.recording = False  
   self.writer = None  
   def run(self):  
   # Open the webcam  
   self.cap = cv2.VideoCapture(0)  
   if not self.cap.isOpened():  
   print("Webcam thread: camera not opened")  
   return  
   self.is\_running = True  
   # (Optional) set resolution or other properties here if desired  
   # e.g., self.cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 1280)  
   # self.cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 720)  
   while self.is\_running:  
   ret, frame = self.cap.read()  
   if not ret:  
   break # camera error or end  
   # If recording, write frame to file  
   if self.recording and self.writer:  
   self.writer.write(frame)  
   # Convert frame to QImage for GUI  
   image = self.convert\_frame\_to\_qimage(frame)  
   # Emit the frame for GUI thread to update  
   self.frame\_signal.emit(image)  
   # Throttle the loop to reduce CPU (optional, e.g., 30 FPS max)  
   time.sleep(0.03) # ~33ms for ~30fps; adjust as needed  
   # Clean up when loop ends  
   self.cap.release()  
   if self.writer:  
   self.writer.release()  
   print("Webcam thread: stopped.")  
    
   def stop(self):  
   """Stop the thread and wait for it to finish."""  
   self.is\_running = False  
   # QThread quit() can be used, or just exit loop. We can also call:  
   # self.wait() to block until thread finishes, if needed.  
    
   def start\_recording(self, filepath, fps=20.0):  
   """Start recording to the given file (initialize VideoWriter)."""  
   # Define codec and create VideoWriter  
   fourcc = cv2.VideoWriter\_fourcc(\*'XVID') # or \*'MJPG'  
   width = int(self.cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))  
   height = int(self.cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))  
   self.writer = cv2.VideoWriter(filepath, fourcc, fps, (width, height))  
   if not self.writer.isOpened():  
   print("Error: VideoWriter failed to open.")  
   else:  
   self.recording = True  
   print(f"Webcam recording started: {filepath}")  
    
   def stop\_recording(self):  
   """Stop recording and release the video writer."""  
   self.recording = False  
   if self.writer:  
   self.writer.release()  
   self.writer = None  
   print("Webcam recording stopped.")  
    
   def convert\_frame\_to\_qimage(self, frame):  
   """Convert a BGR frame (numpy array) to QImage for display."""  
   from PyQt5.QtGui import QImage  
   rgb\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB) # OpenCV uses BGR, convert to RGB  
   h, w, ch = rgb\_frame.shape  
   bytes\_per\_line = ch \* w  
   # Create QImage from numpy data (Format\_RGB888 for 3-channel RGB)  
   qt\_image = QImage(rgb\_frame.data, w, h, bytes\_per\_line, QImage.Format\_RGB888)  
   return qt\_image
* This is a **skeletal implementation**. Let’s highlight what’s happening:

1. frame\_signal: a PyQt signal that will carry the frame (as a QImage or QPixmap) to the main thread. We use object as type for generality (could also specify QImage if we import it in advance).
2. run(): This is the QThread’s entry point when start() is called. We open the webcam and enter a loop reading frames. For each frame:
   * If recording is active (self.recording == True), we write the frame to the video file via self.writer.write(frame).
   * We always convert the frame for display and emit it. We’ll define convert\_frame\_to\_qimage() to handle the conversion.
   * We include a small time.sleep() to limit the frame rate (to ~30 fps in this example). This prevents the loop from running at the maximum possible speed which could be CPU-intensive. Adjust the sleep or implement a frame timer if more precise control is needed. For instance, 0.03s sleep ~ 33 fps max; using 0.05s would limit to ~20 fps. You could also measure cv2.CAP\_PROP\_FPS and use that as target frame rate.
3. stop(): This method will allow us to stop the loop gracefully by setting is\_running to False. We could call this when the application is closing to ensure the thread ends and releases the camera. After setting the flag, the loop will break on next iteration and clean up.
4. start\_recording(filepath): Initializes a cv2.VideoWriter to start saving frames. We use a codec FourCC like XVID or MJPG. (More on codecs below.) We fetch the frame width and height from the capture to ensure the writer uses the same resolution. We also pass an FPS value; 20.0 or 30.0 are typical. Once the writer is open, we set self.recording = True so that the run loop will start writing frames. We print a debug message for confirmation.
5. stop\_recording(): Stops writing by resetting the flag and releasing the writer. After this, frames will no longer be written to file (though preview continues). We also print a debug message.
6. convert\_frame\_to\_qimage(frame): Helper to convert an OpenCV BGR frame (NumPy array) to a QImage. We convert BGR to RGB (since QImage expects RGB byte order for Format\_RGB888). Then we create a QImage using the raw data buffer. We can directly use QImage(frame.data, width, height, bytes\_per\_line, Format\_RGB888) which shares memory; this is fine as we emit it immediately. Alternatively, to be safe, we might copy the data (QImage.copy()) if there’s a concern about scope, but emitting as is should be okay if handled quickly by the main thread. We return the QImage. (We could also convert to QPixmap here itself, but it’s common to emit QImage and convert to pixmap in the main thread slot.)

This thread class uses OpenCV to grab frames and PyQt signals to communicate. **Using QThread in this way ensures the GUI stays responsive** while the camera runs[[1]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=Since%20reading%20the%20camera%20feedback,the%20GUI%20of%20our%20application). Also, separating capture logic here makes our design modular.

1. **Frame Conversion Rationale:** As noted, OpenCV gives us frames as NumPy arrays (in BGR color). To display these in a PyQt QLabel, we must convert them to a QImage (RGB) that Qt understands[[2]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=When%20OpenCV%20read%20a%20frame,that%20will%20perform%20this%20conversion). We’ve implemented that in convert\_frame\_to\_qimage(). This conversion is relatively fast, but it does add some overhead each frame. We use Format\_RGB888 which corresponds to 24-bit RGB. This is compatible with the typical formats of webcam frames. (If needed, one could explore Format\_RGB32 if aligning to 32-bit words, but it’s not necessary here.)
2. **Limiting Frame Rate:** The loop uses time.sleep() to throttle. Another approach is using a Qt timer to trigger capture at intervals, but since we already have a loop, sleep is straightforward. The actual camera might be providing 30fps; if our sleep is a bit off, it’s okay. This is just to prevent 100% CPU usage. We can refine this later if needed (for example, calculating elapsed time or using QElapsedTimer for more precise control). For now, a rough limit of ~20-30 fps is sufficient and reduces resource hogging as planned.
3. **Alternate Approach – QTimer in Main Thread:** Just for context, an alternative would be to not use a separate thread at all, and instead use a QTimer in the GUI thread to periodically grab frames (e.g., every 50 ms). This can work for low frame rates, but if grabbing frames or encoding takes some time, it can still stutter the UI. Using a QThread is a cleaner approach for continuous capture, as we’ve done.

**Testing Checkpoint – Webcam Thread:** At this stage, you can test the thread in isolation. For example, create an instance of WebcamCaptureThread in a simple PyQt app, connect its frame\_signal to a slot that updates an image, and start it. We will do exactly that in the next step when integrating with the GUI. If you run the thread (without even recording), it should continuously print frames via the signal or any debug inside the loop. Ensure no errors occur. If the thread stops immediately, check that the camera is not already in use and that cap.read() returns frames.

Now that the thread class is ready, let’s integrate it into the main application’s GUI.

## Step 3: Integrating Webcam Preview into the GUI

**Goal:** Add a live preview panel in the PC app’s interface to display the webcam feed. We will update the main window to create a WebcamCaptureThread instance and show its frames on a QLabel.

1. **GUI Layout – Adding a Preview Widget:** If your PC app already has a GUI (likely with controls for connecting to phones, starting sessions, etc.), decide where the webcam preview will be displayed. A simple approach is to add a QLabel in the main window dedicated to the webcam video. For example, you might have a section in the UI labeled “PC Camera Preview” containing the QLabel. If you are using Qt Designer for the UI, you can drop a QLabel widget into the layout and give it a name (e.g., labelPcCamera). If creating UI in code, instantiate a QLabel and add to a layout or as a central widget component.
2. **Size and Scaling:** Set an appropriate size for the QLabel. It could be a smaller thumbnail or a larger view depending on your UI. You may use QLabel.setFixedSize() or allow it to scale. Typically, you might set a preferred size like 640x480 for the preview. You can also enable scaling by doing label.setScaledContents(True) so that the pixmap scales with the label size (or manually scale the QImage as we did in the thread with scaled() if needed). In our thread code above, we did not resize the frame (except converting color), so it will emit full-size frames. If your webcam is HD, that could be a large pixmap; for performance, you might scale it down when converting to QImage (e.g., use OpenCV cv2.resize or QImage.scaled). This is optional – if performance is fine, you can display full resolution in a resizable label.
3. **Initializing the Thread in Main Window:** In your main window class (e.g., MainWindow or similar QMainWindow subclass), create an instance of WebcamCaptureThread. You might do this in the \_\_init\_\_ after setting up UI elements. For example:

* self.webcam\_thread = WebcamCaptureThread()
* Connect the thread’s signal to a slot method in the main window that will handle incoming frames:
* self.webcam\_thread.frame\_signal.connect(self.update\_webcam\_frame)
* Here, update\_webcam\_frame is a method we will define to receive the QImage and set it on the QLabel.

1. **Slot to Update QLabel:** Define a slot method in the main window class to accept the frame signal. For example:

* from PyQt5.QtGui import QPixmap, QImage  
  # inside MainWindow class:  
  @QtCore.pyqtSlot(object)  
  def update\_webcam\_frame(self, image: QImage):  
   """Slot to receive QImage from webcam thread and update the QLabel."""  
   # Convert QImage to QPixmap and set it on the label  
   self.labelPcCamera.setPixmap(QPixmap.fromImage(image))
* We use @pyqtSlot(object) to declare the slot (optional but good practice). The slot simply takes the QImage (or object) emitted, converts to QPixmap, and sets it on the label. We assume the QLabel for webcam is accessible as self.labelPcCamera (either from Qt Designer or created in code). This will instantly update the GUI with the new frame. Qt signals/slots ensure this runs in the GUI thread, so it’s thread-safe to update the widget.

1. **Start the Webcam Thread:** Decide when to start capturing. Options:
2. **Automatic Start:** Start the thread when the application launches (or when the main window is shown). This means the webcam feed is always running in preview. You could put self.webcam\_thread.start() at the end of your main window initialization. This way, as soon as the app opens, the webcam preview begins.
3. **On-Demand Start:** If you prefer not to have the webcam on constantly (for privacy or performance), you can control it with a button (like “Enable Webcam Preview”). The Medium example we referenced used a button to start the camera[[3]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=self.open_btn%20%3D%20QtWidgets.QPushButton%28,open_btn). For simplicity, you might opt to start it immediately, since the user presumably opened the app to record anyway. If you add a toggle, then the user must click it to see the preview.

For now, let’s assume **automatic start** of preview. So in MainWindow.\_\_init\_\_ or a suitable place, add:

self.webcam\_thread.start()

This will invoke WebcamCaptureThread.run() in a new thread, and frames should begin flowing to your update\_webcam\_frame slot.

1. **Error Handling:** If the webcam fails to open, our thread’s run will print an error and return. You might want to handle that in the main GUI – e.g., if after starting the thread you can check if not self.webcam\_thread.isRunning(): or better, have the thread emit an error signal if camera can’t open. This way you could notify the user “Webcam not found.” This is an enhancement; at minimum, check console logs.
2. **Multiple Previews Consideration:** If earlier milestones included previews from the phones (e.g., if the phones stream a low-res feed for monitoring), you might already have a UI area for device previews. If so, you can integrate the PC camera preview similarly, perhaps as another “device” in a list or a dedicated fixed panel. The architecture could treat the PC webcam as just another camera source. In code, you could even manage all camera previews through a unified interface (though unless you implemented phone streaming, this might not apply). For now, we simply add the PC preview.

**Testing Checkpoint – GUI Preview:** Run the PC application now. You should see the GUI come up and within a second or two, the webcam feed should appear in the designated QLabel. Verify the following:

* The **preview is visible** and updating (you should see motion as you move in front of the camera).
* The GUI remains responsive (you can click other buttons or menus without lag). The separate thread ensures this (if you notice UI freezing, something might be wrong in thread usage).
* **Frame rate/latency:** There might be a slight delay (a few tens of milliseconds) due to threading and conversion, but the preview should be close to real-time. If the video is very choppy or slow, consider reducing resolution or check if the throttling delay is too high (or remove the sleep to see max rate, though that may max out CPU). For a typical 720p webcam, a modern PC should handle 20-30fps easily with this setup.
* **CPU usage:** During preview (not recording), CPU usage should be moderate. If it’s using a full core, consider tuning (smaller frames or more sleep). But if performance is acceptable, proceed.

Now that live preview is working, the next step is to enable recording of this webcam feed to a file during sessions.

## Step 4: Enabling Webcam Video Recording on the PC

**Goal:** Save the webcam video to a file when a recording session is started, and stop/save the file when session ends. We’ll leverage OpenCV’s video writing capabilities (cv2.VideoWriter) in the webcam thread.

1. **Using cv2.VideoWriter:** OpenCV provides VideoWriter to encode and save video frames to a file. We already included a simple implementation in our thread (start\_recording method). To reiterate:
2. Choose a **video codec** via FourCC code. FourCC is a 4-byte code identifying the codec (e.g., XVID, MJPG, H264, etc.)[[4]](https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/#:~:text=Firstly%2C%20we%20specify%20the%20fourcc,XVID%27%29%20for%20DIVX). Some common choices:
   * 'XVID': Often used on Windows to produce an AVI file with MPEG-4 Part 2 codec (editable by many players). This is a safe default on Windows[[4]](https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/#:~:text=Firstly%2C%20we%20specify%20the%20fourcc,XVID%27%29%20for%20DIVX).
   * 'MJPG': Motion-JPEG, produces relatively large files but very widely compatible. Good if other codecs aren’t available.
   * 'MP4V': A variant for MPEG-4, often used with .mp4 files.
   * 'X264' (H.264) might produce smaller mp4 files but typically requires OpenCV to be built with proper codec support (OpenCV’s FFmpeg backend).
3. **File format:** Commonly **.avi** with XVID or MJPG on Windows (since AVI is straightforward and doesn’t require advanced codecs). If using MP4 container (e.g., .mp4), ensure the codec is compatible (like MP4V or H264). For now, we can use AVI with XVID for simplicity.
4. **Frame size:** Must match the frames you write. Use the capture’s width and height (as we did).
5. **Frame rate:** Set an appropriate fps for the writer. It should roughly match the actual capture rate. If unsure, 20.0 or 30.0 are fine. If the webcam can’t actually deliver that, the video might have duplicated frames, but that’s okay. Alternatively, use cap.get(cv2.CAP\_PROP\_FPS) if it returns a reasonable value (sometimes it may be 0 or unsupported, depending on camera/OS).

Example initialization (from our thread code):

fourcc = cv2.VideoWriter\_fourcc(\*'XVID')  
self.writer = cv2.VideoWriter('session123\_pc\_cam.avi', fourcc, 20.0, (width, height))

Always check self.writer.isOpened() to ensure the file opened properly. If it returns False, the codec or file path might be an issue.

1. **Starting the Recording:** The actual trigger to start recording will come from the user action (e.g., clicking “Start Recording” in the GUI). We will integrate with session management in the next step, but essentially, when the session starts:
2. Determine the output file path for the PC video (more on naming in Session step).
3. Call webcam\_thread.start\_recording(file\_path, fps) to begin writing.
4. The thread will then start writing each frame in its run loop to that file until told to stop.

In the GUI code, if you have a method like start\_session() handling the Start Recording button, you would add something like:

# inside start\_session or on\_start\_button\_clicked:  
session\_folder = create\_session\_folder(...) # ensure directory exists  
pc\_video\_path = os.path.join(session\_folder, "PC\_webcam.avi")  
# Or include session name or timestamp in file name as desired  
self.webcam\_thread.start\_recording(pc\_video\_path, fps=20.0)

We will cover folder creation in the session step, but the idea is straightforward.

1. **Stopping the Recording:** When the user clicks “Stop Recording” (ending the session):
2. Call webcam\_thread.stop\_recording(). This will flush and close the video file. It’s important to do this before closing the app or starting a new session, to ensure the file is finalized (headers written etc.). Our thread’s method releases the writer which finalizes the AVI file.
3. You might also wait a brief moment to ensure the last frame is written, but releasing is usually enough.

In GUI stop handler, for example:

self.webcam\_thread.stop\_recording()  
# (Then you might proceed to collect phone videos or whatever is next)

1. **Codec Considerations:** As mentioned, OpenCV on different OS has different available codecs[[4]](https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/#:~:text=Firstly%2C%20we%20specify%20the%20fourcc,XVID%27%29%20for%20DIVX). The Python opencv-python package comes with its own FFmpeg, which typically supports many codecs. On Windows, 'XVID' should work (producing an AVI that uses the MPEG-4 codec, broadly playable). 'MJPG' will produce large AVIs but guaranteed to work. If you prefer MP4 output, you can try:

* fourcc = cv2.VideoWriter\_fourcc(\*'mp4v')  
  self.writer = cv2.VideoWriter('pc\_video.mp4', fourcc, 20.0, (width, height))
* Many have success with 'mp4v' or 'X264' for MP4 files, but it may require the ffmpeg binaries. If you find the output file not playing, it could be codec issues – in that case, fall back to AVI/MJPG. You can refine codec settings later or use external tools for conversion. For now, the priority is to **get a working recording**.

1. **Recording Quality and Performance:** Initial implementation might use uncompressed or lightly compressed video (like MJPEG). This results in large files for long recordings. If that becomes an issue (say, recordings are an hour long – MJPEG AVIs would be huge), consider integrating a more efficient encoding later (H.264). That could be done by calling an ffmpeg subprocess to encode the raw frames, or by ensuring OpenCV’s FFmpeg supports H264. This is an advanced improvement. At Milestone 3.3, we accept larger files in exchange for simpler implementation. We can note this as a future enhancement.
2. **Audio (Out of Scope):** We are only capturing video. The PC webcam’s microphone (if any) is not captured here. The phones might be capturing audio (if that was a requirement), but the PC’s audio is separate. Capturing audio via OpenCV is not supported; PyQt’s QMediaRecorder could, or using pyaudio/ffmpeg. If audio syncing is needed, that’s another complex topic, likely beyond current scope. For now, we assume only video streams are needed (as per the prompt focusing on video capture).
3. **Time Synchronization:** When starting the recording, record the PC’s system time (timestamp). This can be used to align with phone videos later. For example, you could do:

* import datetime  
  start\_time = datetime.datetime.now()  
  print(f"Session started at {start\_time.isoformat()}")
* or log it to a file in the session folder. Because the PC is the one triggering the session, you can assume the phone recordings start nearly at this time (network delays maybe a few milliseconds). If high precision sync is needed, one could send a sync signal or use a common clock reference, but that’s beyond our current scope. At least noting the start time in a log file (like session\_start\_time.txt) is useful.

Now the PC is capable of recording its webcam. We will integrate this with the broader session workflow (creating folders, coordinating with phone recordings).

**Testing Checkpoint – PC Recording:** It’s a good idea to test PC webcam recording alone first: you can simulate a session start manually. For example, run the app, let the preview come up, then in the console or via a temporary button, call webcam\_thread.start\_recording("test.avi", fps=20.0). Let it record 5-10 seconds, then call webcam\_thread.stop\_recording(). Check that **test.avi is created** in your working directory (or specified path). Open it with a video player to ensure it’s not corrupted. You should see the video of those few seconds. Verify the content matches what was in the preview. This confirms that the VideoWriter is working properly. If the file is not playable or empty, double-check codec and writer initialization. (A common mistake is a wrong frame size or forgetting to release the writer – ensure writer.release() is called on stop, otherwise the file may not finalize properly.)

With a confirmed working recording, proceed to tie it into session management.

## Step 5: Session Management and Multi-Device Coordination

**Goal:** Incorporate the webcam recording into the overall session start/stop so that all devices (phones and PC) record together, and organize the outputs neatly.

1. **Session Concept:** A “session” represents one recording event (e.g., a trial in an experiment, or a scene in data collection). We will create a session folder and store all related files in it. For instance, if the session is identified by a timestamp or name, use that in the folder name and file names. Example structure:

* Recordings/  
   Session\_2025-07-27\_17-00-00/  
   PC\_webcam.avi  
   phone1\_video.mp4 (files from phone devices)  
   phone2\_video.mp4  
   session\_log.txt (optional log of events/timestamps)
* This keeps things organized.

1. **Generating Session Name/ID:** You can allow the user to input a name (via a QLineEdit in the UI). Or automatically generate one using the current date-time. A safe approach is to create a timestamp string when “Start Recording” is pressed, especially if the user doesn’t provide a name. For example:

* import datetime, os  
  base\_dir = "C:/Recordings" # base directory for all sessions (could make this configurable)  
  timestamp = datetime.datetime.now().strftime("%Y-%m-%d\_%H-%M-%S")  
  session\_name = f"Session\_{timestamp}"  
  session\_path = os.path.join(base\_dir, session\_name)  
  os.makedirs(session\_path, exist\_ok=True)
* Make sure base\_dir exists or create it as well if needed. Use a consistent format for the folder name to sort chronologically.

1. **Starting a Session (in GUI logic):** When the user clicks “Start Recording”:
2. Optionally gather a session name from UI (or just use timestamp as above).
3. Create the session folder.
4. **Trigger Phone Recordings:** (Assuming previous milestones have established a connection to the phones, likely via network commands or ADB, etc.) You would send a message/command to each connected phone to begin recording. For example, if using a TCP or UDP command, something like {"action": "START\_RECORD", "session": session\_name} could be sent. The phones then start their camera recording (and possibly send back confirmation). This part is outside the PC code’s direct scope except for sending the command. Ensure that is non-blocking (likely it is quick).
5. **Start PC Recording:** Immediately call self.webcam\_thread.start\_recording(pc\_video\_path, fps). The pc\_video\_path should be within the session folder, e.g., session\_path + "/PC\_webcam.avi". If you have multiple PC camera views (not in this case), name accordingly. We have just one, so “PC\_webcam” or simply use session name + “\_PC.avi”.
6. **UI Feedback:** Change the UI state to indicate recording is in progress. For example, disable the Start button (or change it to a Stop button), display the session name somewhere, maybe show a red dot or “Recording...” label. These help the user know the recording is live. Also handle any errors (if a phone didn’t respond, etc., though ideally you handle that prior or have timeouts).
7. **Stopping a Session:** When “Stop Recording” is clicked:
8. Send command to phones to stop recording. They should stop and possibly save their files. In some setups, the phones might then upload the file to the PC or wait for the PC to fetch it. (If that’s in scope, you might have to implement file transfer via network or USB. If not, the phone videos remain on devices and you manually gather them later. The requirement isn’t explicit here, but a complete platform might retrieve them. We’ll focus on PC side.)
9. Stop the PC webcam recording: call self.webcam\_thread.stop\_recording(). This closes the PC video file.
10. Mark the session as ended. UI-wise, you can re-enable the Start button for a new session, show a “Recording saved” message, etc. Perhaps list the saved files or duration.
11. If any post-processing is needed (like copying phone files if they were streaming or uploading), handle that. If not, at least you can log that all devices have stopped.
12. **Logging Session Events:** It’s useful to create a small log of what happened during the session, for future reference or debugging. You can write a text file in the session folder, e.g., session\_log.txt, containing info like:

* Session: Session\_2025-07-27\_17-00-00  
  Start Time (PC clock): 2025-07-27 17:00:00.123456  
  PC Video: PC\_webcam.avi  
  Phone1 video: phone1\_2025-07-27\_17-00-00.mp4  
  Phone2 video: phone2\_2025-07-27\_17-00-00.mp4  
  ...  
  Stop Time (PC clock): 2025-07-27 17:00:10.987654  
  Duration: ~10.8 seconds
* This kind of log can be generated if needed. At minimum, log the start time as noted earlier. This is helpful for later analysis (especially to align with any external data or simply to know when it was recorded).

1. **Synchronizing Time:** Note that each device (PC and phones) has its own internal clock for timestamps. If perfect synchronization is required, ensure all devices’ clocks are reasonably in sync (perhaps via NTP). However, since the PC initiates the start, the relative delay is what matters. Likely, the phone apps were waiting for the start command and start recording immediately upon receipt. Network latency could cause a few hundred milliseconds of difference at worst (depending on implementation). In many scenarios, this is acceptable. If not, one might implement a countdown or sync signal (for example, PC sends a sync pulse or audio clap that all cameras capture). But discussing those methods is beyond this milestone’s scope. For now, assume the simultaneous start is sufficient for sync “good enough”.
2. **Treating PC as a Device (Design Perspective):** As a design improvement, you could model the PC’s own camera as just another “Device” in your system. For example, if you have a list of CameraDevice objects (for phones), you could also have a PcCameraDevice object that implements start/stop. That object would wrap around the WebcamCaptureThread. This way, your Session manager can loop through all devices (phones + PC camera) and call a generic .start\_recording() on each. This isn’t strictly necessary, but it can make the design elegant. If you prefer not to abstract that far, just handle the PC separately as we have (since it’s local and different from remote devices).

Now everything is set for a full test of the integrated system.

## Step 6: Testing and Verification of Webcam Integration

This is a critical milestone; thorough testing is needed to ensure reliability. Go through the following tests and checkpoints:

**A. Basic Webcam Preview Test:**  
- Launch the PC application. Confirm that the **webcam preview is running** in the UI (as done in Step 3 testing). If it’s not, re-check thread start logic or camera index.  
- Try resizing the window (if allowed) to see how the preview scales. Ensure it doesn’t crash or behave oddly. Minor aspect ratio stretching is okay if scaled contents; you can refine to preserve aspect ratio if needed (e.g., by adjusting label or pixmap scaling).

**B. Start/Stop Recording (PC alone):**  
1. Click the **Start Recording** button in the PC app to initiate a session.  
2. Observe the UI: it should indicate recording started (e.g., button toggled to “Stop”, a recording timer if implemented, etc.).  
3. Let it record for a short duration (say 5-10 seconds) while moving a bit in front of the camera (to have some motion in video).  
4. Click **Stop Recording**. The UI should update (button back to “Start”, etc.). No errors/exceptions should occur during this process.  
5. Now navigate to the recordings output directory (e.g., C:\Recordings\Session\_2025-07-27\_17-00-00\). Confirm that a folder was created for the session (with the correct timestamp/name). Inside it, find the PC\_webcam.avi (or chosen name) file.  
6. Play the video file using a standard media player (Windows Media Player, VLC, etc.). **Verify the video plays correctly**: you should see the webcam footage you just recorded, with the expected duration. Check that the video is not empty or corrupted (if it doesn’t play, try VLC which supports raw AVI better; if still an issue, likely a codec problem – consider switching codec as discussed).  
7. (Optional) If you have access to the phone videos for that session, check their existence too. Depending on system design, the phone might have saved its video either on phone storage or sent it to PC. Verify phone recordings are indeed happening (maybe a message on phone app or an indicator in PC app if implemented).

**C. Multi-Device Sync Test:**  
This test checks if the PC and phone(s) recordings truly start together:  
1. Set up the phones and PC to record a common event. For instance, position the phone cameras and PC camera toward a common scene or a **sync signal**. A classic sync test is to do a clap or use a flashing LED that all cameras can see.  
2. Start a session via the PC app. Do a hand clap in view of all cameras (including the PC webcam) at a known time shortly after start. Stop the recording after a few more seconds.  
3. Retrieve the phone videos (if not automatically transferred, you might manually copy them from the device for this test).  
4. Compare the footage: The clap should be visible/audible. Check the **timestamp or frame index** of the clap in each video: - If you have no timestamp overlay, you can just play them side by side or scrub to the clap moment. You should find that the clap occurs roughly at the same offset from the start in all videos. There might be a slight difference (e.g., clap at 2.0s in PC video, 1.8s in phone1, 2.1s in phone2). Small differences are expected due to trigger and camera internals. They can be adjusted in post-processing if needed. - If the differences are large (say one video starts much later), there may be a sync issue in sending the start command or a delay in a device. Investigate the phone app’s response time in that case. The PC video’s start time is essentially immediate on button press; phone might have a fraction of a second delay. Usually this is fine, but just validate it’s not more than, say, 0.5s unless network lag is high. - If needed, adjust approach (for example, send a countdown to start or confirm all devices ready before recording). However, for now, just note the sync accuracy.

1. During this test, also monitor if any frames were dropped or if either video has unexpected pauses. If the PC’s video shows no issues but phone videos have gaps, the issue lies with phone side (outside this milestone’s scope). If the PC video had any problem, check the PC performance.

**D. Performance and Stability:**  
- Run a **longer recording test**: e.g., let the system record for several minutes. Ensure the PC app doesn’t run out of memory or crash. The thread should handle continuous writing. Check that file size grows appropriately and no errors occur over time.  
- Observe CPU and memory: The PC app (Python) might use some CPU for encoding (especially if using MJPG, which is actually quite CPU-light since it’s just JPEG each frame). Ensure it’s within acceptable range. If you notice very high CPU usage, consider lowering frame rate or resolution. For example, if not already done, setting the webcam to 720p instead of 1080p can cut down processing. Or explicitly skip frames (e.g., only process every nth frame for preview if you only need, say, 15 fps preview but still record full – though recording full and previewing partial complicates things, so probably keep them same). - **Stopping and Starting Multiple Times:** Do multiple session recordings back-to-back: 1. Start session 1, record 5 seconds, stop. 2. Start session 2, record a few seconds, stop. 3. Ensure a new folder was created for session 2, and that the webcam thread successfully closed the first file and started a new one. Check that the second file is valid. - This tests that our start\_recording after a previous stop\_recording works. We should have released the writer, so it should be fine. If for some reason the second video file is empty or the thread crashed, there might be a bug in how we reinitialize the writer or flags (double-check that after stopping, self.recording is False and writer is None before starting again). - Verify no resource leak: after stopping a session, the webcam preview should still continue (since we keep the thread running for preview). If you choose, you can leave the thread running indefinitely, which is simplest. Alternatively, you could stop the thread entirely after each session if you wanted to close the camera, but that adds overhead of reopening for next session. It’s usually fine to keep it open if frequent sessions are expected. However, if there’s a long gap between sessions and you want to free the camera, you could implement that (not required now).

**E. Robustness:**  
- Try unplugging/disabling the webcam (if external) and see how the application handles it. It should ideally report an error and not crash. This might be beyond normal use (the user presumably ensures a camera is present), but it’s good to know failure modes. - If the app might be used on PCs without a webcam, ensure that scenario is handled (either disable the preview/record feature or show “No camera”). Perhaps have the Start Recording button still function but just not record PC video if no webcam. Since the prompt expected webcam support, likely a webcam will be there.

**F. UI/UX considerations:**  
- Ensure the added webcam preview and recording features are clearly presented to the user. For example, label the preview “PC Camera” so they know what it is showing. - If the user should have an option to turn off PC recording (maybe they only want phones sometimes), you could provide a checkbox like “Include PC webcam in recordings”. If unchecked, the PC won’t record (you could simply not start the writer). But default might be to always include it. - Double-check that closing the application stops the thread properly. If the app is closed while a session is ongoing, you should handle that (stop recording, close files). On window close, call webcam\_thread.stop() to break the loop, then maybe webcam\_thread.wait() to ensure it fully exits. This prevents the Python process from hanging due to a leftover thread. In PyQt, if threads are not stopped, the app might hang on exit. So implement the closeEvent in MainWindow:

def closeEvent(self, event):  
 # Stop webcam thread  
 if self.webcam\_thread.isRunning():  
 self.webcam\_thread.stop()  
 self.webcam\_thread.wait(1000) # wait up to 1 second for it to terminate  
 event.accept()

This will ensure graceful shutdown. (Alternatively, set the thread as daemon or use self.webcam\_thread.quit(), but since we wrote our own loop, setting is\_running=False is fine.)

By completing these tests, you confirm that **Milestone 3.3** is successfully implemented: the PC app now captures and records its webcam video in sync with the smartphones, and everything is packaged in a session format.

## Additional Tips and Considerations

* **IDE Tips:** During development in an IDE, it can be helpful to run the GUI in debug mode to catch any exceptions. If the GUI window closes immediately, look for errors in thread usage (common issues are calling GUI elements from the thread – but we correctly used signals to avoid that). Make use of logging or print statements as we did for start/stop recording to trace the flow in the console.
* **PyQt Threads Best Practice:** We used a subclass of QThread. Another pattern is to use moveToThread with a worker object, but for this use-case, subclassing is straightforward. We must be careful not to interact with widgets from within the thread (we didn’t – we only emit signals). Also note, we called self.terminate() in the AranaCorp example’s stop method[[5]](https://www.aranacorp.com/en/displaying-an-opencv-image-in-a-pyqt-interface/#:~:text=def%20stop%28self%29%3A%20self) – that’s generally not recommended unless the thread is truly stuck, as it forces termination. We did not use terminate() in our code; we rely on the loop exit. This is safer. We only call quit() or wait() from the main thread when stopping.
* **Resource Cleanup:** If the user starts a session and then closes the app without clicking Stop, try to handle that too. The closeEvent logic above helps. Also, consider if phone recordings need a stop signal on app close as well (to avoid them recording indefinitely).
* **Extending to Multiple PC Cameras:** If the project ever requires capturing more than one webcam on the PC (for example, maybe a front and a secondary camera), you could replicate this module for multiple cameras (with different indices). You’d need multiple threads and multiple preview labels. The design we used scales to that (just create additional threads and labels).
* **Future Improvement – FFMPEG for Recording:** If file size or codec efficiency becomes a priority, you might integrate FFmpeg directly. One approach: instead of writing frames via OpenCV’s writer, pipe raw frames to an ffmpeg process (spawned with subprocess) that encodes to H.264 on the fly. This can produce smaller mp4 files. However, this adds complexity and potential issues (ensuring ffmpeg is installed, handling the pipe). For now, our approach should suffice and you can post-convert if needed.
* **User Confirmation:** After recording, you might present a quick summary to the user, like “Saved PC video: 10.5 MB, 10 seconds” or simply list file paths, so they know it succeeded. This can be done via a message box or status bar update. It’s optional but improves user feedback.
* **Ensure Qt Event Loop Runs:** One subtle point: when the PC is recording, the thread does heavy work but the main GUI must remain active to process events (like the Stop button click). As long as we don’t hog the main thread, it will. Our design is good in that regard. Just be mindful that doing too much on the main thread (like waiting for phone file transfers synchronously) could freeze the UI. If, for example, after stop you copy large files from phones, consider doing those in background threads or show a progress bar. But again, that’s beyond the webcam focus.

With all the above implemented and tested, **Milestone 3.3 is achieved**. The PC application now robustly handles its own camera: capturing live video, displaying a preview, and recording to file in sync with the other devices. This enhances the data collection platform by providing a PC-recorded video stream (e.g., of the user or the overall scene) to complement the mobile device recordings. You can now proceed to the next milestones (if any), such as perhaps combining all streams, post-processing, or adding other sensor data streams to the recordings. Congratulations on extending the system’s functionality with PC webcam integration![[1]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=Since%20reading%20the%20camera%20feedback,the%20GUI%20of%20our%20application)[[4]](https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/#:~:text=Firstly%2C%20we%20specify%20the%20fourcc,XVID%27%29%20for%20DIVX)

[[1]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=Since%20reading%20the%20camera%20feedback,the%20GUI%20of%20our%20application) [[2]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=When%20OpenCV%20read%20a%20frame,that%20will%20perform%20this%20conversion) [[3]](https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7#:~:text=self.open_btn%20%3D%20QtWidgets.QPushButton%28,open_btn) Display OpenCv camera on a PyQt app | by Baadji ilias | Medium

<https://medium.com/@ilias.info.tel/display-opencv-camera-on-a-pyqt-app-4465398546f7>

[[4]](https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/#:~:text=Firstly%2C%20we%20specify%20the%20fourcc,XVID%27%29%20for%20DIVX) Saving Operated Video from a webcam using OpenCV - GeeksforGeeks

<https://www.geeksforgeeks.org/python/saving-operated-video-from-a-webcam-using-opencv/>

[[5]](https://www.aranacorp.com/en/displaying-an-opencv-image-in-a-pyqt-interface/#:~:text=def%20stop%28self%29%3A%20self) Displaying an OpenCV Image in a PyQt interface • AranaCorp

<https://www.aranacorp.com/en/displaying-an-opencv-image-in-a-pyqt-interface/>