# Milestone 3.5: Stimulus Presentation Controller – Technical Implementation Guide

## Goal and Overview of Stimulus Presentation

The goal of Milestone 3.5 is to enable the PC application to present visual stimuli (such as video files) in sync with the ongoing multi-device recording session. In a typical experiment scenario, this means playing a video on the computer screen (or a connected display) for the participant while simultaneously recording data from multiple devices (e.g. smartphones, PC webcam). We need to ensure that the system captures accurate timing information for the stimulus presentation, so that recorded data can be synchronized with the stimulus timeline during analysis.

By adding this **Stimulus Presentation Controller** feature, our system evolves from a passive recorder into an interactive experimental setup. The operator will be able to load a video stimulus, play it for the subject (full-screen if needed), and have the recording system automatically log when the stimulus started and when any events of interest occur. This milestone will cover designing the UI controls for stimulus playback, integrating a video player component (using Qt’s multimedia framework), synchronizing playback with the recording start/stop, logging timing information locally on the PC, and adding conveniences like keyboard shortcuts (e.g. **Spacebar** to play/pause, **Esc** to exit full-screen) for ease of use.

## Design and Class/Module Breakdown

To incorporate stimulus presentation, we will introduce new components and extend existing ones in our application architecture. Below is a breakdown of the classes/modules and their roles in this milestone:

* **MainWindow/UI Module:** The main GUI (likely a QMainWindow subclass) will incorporate new UI elements for stimulus control. This includes buttons like "Load Stimulus", "Play", "Pause", and possibly a combined "Start Recording & Play" button. The MainWindow will manage high-level workflow (e.g. when the user clicks "Start Experiment", trigger both recording and stimulus playback). The UI may also include or manage a video display widget where the stimulus is shown (possibly in a separate window or on a second screen).
* **StimulusController (New Class/Module):** This will be responsible for handling all aspects of stimulus media playback. It will utilize Qt Multimedia classes (QMediaPlayer and QVideoWidget) to load and play video files within the GUI. The StimulusController will provide methods to:
* Load a video file (e.g. via a file dialog).
* Control playback (play, pause, stop).
* Display the video either embedded in the UI or in full-screen mode on a chosen display.
* Log timing information (e.g. stimulus start time, current playback time on events).
* Handle user inputs related to the video (such as key presses for play/pause or exiting full-screen).

Internally, StimulusController will own a QMediaPlayer object (for media control) and a QVideoWidget (for video display). It might subclass QVideoWidget or install event filters to capture key events (like Esc or Space) when in full-screen. This class could be a singleton within the app or an attribute of the MainWindow (depending on design). For simplicity, we can implement it as part of the MainWindow class initially, but logically it's a separate concern.

* **RecordingController/Manager (Existing Module):** We assume from previous milestones that there is a component managing the recording of devices (e.g. sending start/stop commands to smartphones, handling PC webcam recording, etc.). This milestone will integrate with that component. Specifically, when starting an experiment that involves a stimulus, the RecordingController should coordinate with StimulusController to start everything in sync. The RecordingController will also likely receive timing markers or metadata from the StimulusController (for example, logging that “stimulus started at T=...”). If no dedicated class exists, the MainWindow might directly orchestrate recording start/stop and stimulus playback.
* **Data Logger (Possible Module or functionality):** We will create a mechanism to log timing data and markers to a file on the PC. This could be a simple logging utility or just done within StimulusController. It will write events like "Experiment started", "Stimulus play started at X", "Marker at Y seconds" to a timestamped log file for later analysis. This log is stored **locally on the PC** (as per requirements, we are not sending these logs to phones or elsewhere).
* **UI Layout:** The UI will be updated to include the stimulus controls. There might be a dedicated panel or toolbar for media playback (with open/play/pause buttons). The video display area could be:
* Embedded in the main window (useful for operator preview or single-screen usage).
* Or a separate window that can be moved to another display and made full-screen (ideal for the participant’s view if using dual monitors).

We may also have an indicator or small console showing log messages (optional), or at least status messages indicating what’s happening (e.g. "Video playing...").

* **Keyboard Shortcuts:** We will implement shortcuts for common actions in the context of stimulus playback:
* **Spacebar**: Toggle play/pause of the video (for quick control without clicking the button).
* **Esc**: Exit full-screen mode on the video display (bringing the video window back to windowed mode or hiding it).
* (Optional) **F**: Toggle full-screen (common in media players), though the user specifically asked for Esc to exit, and presumably a button to enter full-screen or an automatic full-screen on play.

These classes and components will work together as follows: When the user is ready to run an experiment, they load a video file via the StimulusController UI. The video is prepared but not yet playing. The user can then press a single **“Start Recording & Play Stimulus”** button (or two actions in quick succession) which triggers the RecordingController to start all recordings (phones and PC) and simultaneously instructs the StimulusController to start video playback. The StimulusController will log the start time and display the video (full-screen for the subject if needed). During playback, if the operator presses the **“Mark Event”** button (or a hotkey) to indicate an important moment, the StimulusController will log the current video timestamp (and system time) to the log. After the video ends (or the operator stops it), recordings can be stopped and the log closed. This ensures we have a complete record of what was presented and when, relative to the recorded data.

## Setup and Environment Configuration

Before diving into implementation steps, make sure your development environment is prepared for multimedia functionality:

* **Dependencies:** Ensure that **PyQt5** (or PySide2) is installed and that the Qt Multimedia components are available. In PyQt5, the QtMultimedia and QtMultimediaWidgets modules provide QMediaPlayer and QVideoWidget. Install via pip if needed:
* pip install PyQt5
* PyQt5 comes with multimedia support by default. If using PySide2/PySide6, those also include multimedia but names might differ slightly (e.g., QMediaPlayer usage is similar).
* **IDE Configuration:** You can use any Python IDE (PyCharm, VSCode, etc.) for development. No special IDE setup is required beyond having the correct interpreter with PyQt installed. However, for GUI work, it’s helpful to:
* Enable GUI event loop integration in your IDE’s run configuration if needed. (In PyCharm, simply running the script should be fine; in some interactive consoles you might need %gui qt if using IPython).
* If you plan to design the UI with Qt Designer, load the .ui file via PyQt (uic) or PySide tools. In this project, we can create the UI dynamically in code for simplicity. Ensure the IDE is set to the correct working directory if your video files or other assets are relative path dependent.
* For debugging, note that playing video is performance-heavy; stepping through frame-by-frame in a debugger might not be practical. Instead, use logging or breakpoints on control logic if needed.
* **Multimedia Backend Considerations:** Qt’s media player uses the operating system’s multimedia backend. To avoid issues with video playback:
* Test with common video formats (e.g., an MP4 file with H.264 codec) which are typically supported on most OS by default.
* On Windows, Qt uses WMF or DirectShow – ensure a modern Windows version or appropriate codecs are installed (Windows 10+ has built-in support for MP4/H.264). On Linux, Qt uses GStreamer – make sure GStreamer and the needed plugin packages (like gstreamer1.0-libav for H.264) are installed if you encounter playback issues. If a video fails to play or you get a console warning about codecs, you may need to install additional system libraries or convert the video to a supported format.
* We will initially use QMediaPlayer for simplicity. If you later find that QMediaPlayer does not support a needed format or has performance issues, an alternative is to use **VLC via python-vlc**. VLC can handle virtually any codec. However, integrating VLC means controlling an external player instance or embedding it via its own widget, which is more complex. We will proceed with QMediaPlayer, which should suffice for standard use cases.
* **Hardware:** If possible, have a second monitor connected when testing full-screen stimulus display (to simulate the subject’s screen). If not, you can still test full-screen on your single monitor. Also ensure the PC’s resources are adequate (playing a video while recording from multiple cameras can be demanding). Using a short, low-resolution test video initially is wise to verify functionality and performance.

With the environment ready, we can now implement the feature step by step.

## Implementation Steps

Below is a step-by-step guide to implement the Stimulus Presentation Controller in the application:

1. **Add Stimulus Controls to the UI:** Begin by updating the user interface to include controls for loading and playing the stimulus video. In the MainWindow (or your main UI class), add:
2. A **"Load Stimulus"** button. When clicked, this will open a file dialog for the user to choose a video file (e.g., .mp4). Use Qt’s QFileDialog.getOpenFileName to get the file path. For example:

* file\_path, \_ = QFileDialog.getOpenFileName(self, "Select Stimulus Video", "", "Videos (\*.mp4 \*.avi \*.mov)")  
  if file\_path:  
   stimulus\_controller.load\_file(file\_path)
* This calls a method load\_file on our StimulusController (or directly uses QMediaPlayer) to set up the selected video.

1. **Playback buttons**: "Play" and "Pause". These can be separate buttons or a toggle button. Simpler is to have one **Play/Pause** toggle button that switches its label/icon based on the state. However, implementing two buttons is straightforward: clicking "Play" calls mediaPlayer.play(), clicking "Pause" calls mediaPlayer.pause(). (We will also implement Spacebar to toggle play/pause, which is convenient during an experiment.)
2. A **"Start Recording & Play"** button: This is a critical control that initiates the synchronized start. When clicked, it should:
   1. Ensure a stimulus video is loaded and ready.
   2. Trigger the recording start on all devices (e.g., send the start\_record command to phones via network/ADB and start PC webcam recording).
   3. Immediately start the video playback (mediaPlayer.play()).
   4. Log the event (record the timestamp for when the stimulus started relative to experiment start).
3. (Optional) a **"Mark Event"** button: This allows the operator to log a manual marker during the stimulus presentation. For now, just place the button; we will implement its functionality in a later step (logging the current time). The operator can click it whenever something noteworthy happens (or even press a shortcut key) to insert a timestamped marker in the log.

You may arrange these controls on a toolbar or a dedicated panel in the UI. For instance, a horizontal toolbar with [Load] [Play] [Pause] [Start Recording & Play] [Mark Event]. If using Qt Designer, you can add the buttons there; if creating in code, instantiate QPushButton for each, add to a layout or toolbar, and connect their clicked signals to the appropriate slots/methods.

1. **Integrate QMediaPlayer and QVideoWidget:** Next, create the media playback objects using Qt Multimedia. In our StimulusController (or within MainWindow), initialize:
2. A QMediaPlayer instance for video playback. In PyQt5, you should instantiate it with the video surface role, e.g.:

* from PyQt5.QtMultimedia import QMediaPlayer, QMediaContent  
  from PyQt5.QtMultimediaWidgets import QVideoWidget  
  ...  
  self.mediaPlayer = QMediaPlayer(None, QMediaPlayer.VideoSurface)
* This ensures the player is set up for video output[[1]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=class%20VideoPlayer%28QMainWindow%29%3A%20def%20__init__%28self%29%3A%20super%28%29,PyQt5%20Video%20Player)[[2]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=if%20fileName%20%21%3D%20%27%27%3A%20self,fromLocalFile%28fileName).

1. A QVideoWidget to display the video. For example:

* self.videoWidget = QVideoWidget()
* Decide where to place this widget:
  + If you have a dedicated area in your main window (e.g., a placeholder widget), you can replace it or set it as the central widget. For instance, self.setCentralWidget(self.videoWidget) will make the video widget fill the main window’s center[[3]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=layout%20%3D%20QVBoxLayout%28%29%20layout,playButton). But if the main window is used by the operator, you might not want the video taking over the UI completely.
  + Alternatively, **create a separate window** for the video. You can do this by simply not parenting the QVideoWidget to the main window, or by creating a new QWidget/QMainWindow to act as a stimulus window. For now, we can instantiate self.videoWidget = QVideoWidget() without parent and only show it when needed (or parented to main but shown in full-screen on another screen later).

1. Set the media player’s output to the video widget:

* self.mediaPlayer.setVideoOutput(self.videoWidget)
* This links the player and the widget so that video frames will be rendered in the QVideoWidget[[4]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=self).

1. Optionally, set up an audio output if needed. By default, QMediaPlayer will use the default system audio output. If you need to control volume or output device, you can create a QAudioOutput (in Qt6/PySide6) or use QMediaPlayer’s volume methods in Qt5. For simplicity, we will rely on default audio.

With these in place, we have a basic video player integrated into our app. At this point, test that you can load a video and play it: - Implement the load\_file(path) method: create a QMediaContent (for PyQt5) from the file path and call self.mediaPlayer.setMedia(...) with it[[5]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=fileName%2C%20_%20%3D%20QFileDialog.getOpenFileName%28self%2C%20,QDir.homePath). - Then call mediaPlayer.play() either immediately (for a quick test) or via the Play button. - **Checkpoint:** Run the application, click "Load Stimulus" to pick a video file, then click "Play". You should see the video playing in the QVideoWidget (which might be in your main window or a separate window). Verify that the video frames are visible and the application remains responsive (Qt Multimedia runs playback in a separate thread, so the UI should not freeze). Also confirm audio is playing if the video has sound. - Try the Pause button and then Play again, to ensure toggling works (the video should pause/resume accordingly).

1. **Full-Screen Display on Second Monitor (Visual Presentation):** We want the ability to present the stimulus to the participant, likely in full-screen mode (especially if the experimenter’s screen is different from the subject’s screen). To achieve this:
2. **Multi-screen detection:** Check if a second display is connected. In Qt, you can get a list of screens via QApplication.screens() (or QGuiApplication.instance().screens()). For example:

* screens = QGuiApplication.screens()  
  if len(screens) > 1:  
   second\_screen = screens[1]  
  else:  
   second\_screen = screens[0] # if only one screen, will use primary
* We might provide an option for the user to select which screen to use for the stimulus (if multiple are available). This could be a setting or simply default to the second screen.

1. **Displaying full-screen:** We will use the QVideoWidget for the video output. To show it on the desired screen in full-screen mode, do the following when starting playback:

* # Move the video widget to the target screen and show full-screen  
  if use\_second\_screen:  
   geo = second\_screen.geometry()  
   self.videoWidget.setGeometry(geo) # position window on that screen  
  self.videoWidget.setWindowFlag(Qt.FramelessWindowHint, True) # no window border (optional)  
  self.videoWidget.showFullScreen()
* Calling showFullScreen() will make the QVideoWidget cover the entire screen[[6]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=The%20problem%20is%20that%20the,Qt%3A%3AApplicationShortcut). If you have not explicitly moved it to the second screen, it might default to the primary; setting the geometry to the second screen’s geometry ensures it appears there.

1. **Ensure focus for key events:** When QVideoWidget is full-screen, it becomes its own top-level window. We need to make sure it can handle key presses (like Esc to exit). By default, QVideoWidget may not handle Esc, so we will implement it:
   * **Option A (Subclassing):** Create a subclass StimulusVideoWidget(QVideoWidget) and override keyPressEvent. If the key is Qt.Key\_Escape, call self.setFullScreen(False) (which will return it to windowed mode). If the key is Qt.Key\_Space, toggle play/pause. You can also override mouseDoubleClickEvent if you want double-click to toggle full-screen (common behavior).
   * **Option B (Install Shortcuts/Event Filters):** Use QShortcut or an event filter on the video widget. For example, create a QShortcut(QKeySequence(Qt.Key\_Space), videoWidget, toggle\_play\_pause) and another for Escape that calls a slot to exit full-screen. The key is to set the shortcut’s context to the widget or application appropriately. If we attach the shortcut to the video widget itself, it will be active when that widget is focused (which it is, in full-screen)[[7]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=2)[[8]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=layout%20%3D%20QtWidgets,or). For instance:
   * QShortcut(QKeySequence(Qt.Key\_Space), self.videoWidget, self.handle\_toggle\_play)  
     QShortcut(QKeySequence(Qt.Key\_Escape), self.videoWidget, self.handle\_exit\_fullscreen)
   * Here, handle\_toggle\_play would call play or pause on the mediaPlayer depending on state, and handle\_exit\_fullscreen would call videoWidget.setFullScreen(False) or simply videoWidget.hide() if we want to close it. We might also want to simultaneously pause or stop the video when exiting.
   * We will implement Escape to exit full-screen (and possibly also stop the video if the experiment is meant to end). The user specifically requested **Esc to exit** and **Space to pause/play**, which we'll honor via these shortcuts.
2. **Single-screen scenario:** If only one monitor is present, the experimenter might still want the video full-screen. In that case, the QVideoWidget will cover the entire screen (hiding the controls). The operator can use Esc to get out of it. We should make sure that when full-screen, the other controls (on the main window) are not needed until after the video (since they won't be visible). So typically, the operator would start full-screen stimulus, and only after it finishes or is exited would they regain control of the UI.
3. **Testing full-screen:** Connect a second monitor and run the stimulus in full-screen on it. Observe that the video fills the screen and that pressing **Esc** brings it back (or closes the stimulus window). Also test the **Spacebar** while full-screen: the video should pause and resume. (If these keys don’t work, ensure the shortcuts are set on the video widget or that the video widget subclass is handling the events. Remember that in full-screen mode it’s a separate window, so main window shortcuts won’t apply[[7]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=2).)  
   **Checkpoint:** Confirm that full-screen playback on a second screen works smoothly and that you can exit cleanly. If you do not have a second monitor, test full-screen on your main monitor: start playback, the video should cover the screen; press Esc to exit. The application should return to windowed mode, and ideally the video widget should hide or re-integrate into the UI.
4. **Synchronize Stimulus Playback with Recording Start:** A key requirement is that when the stimulus video starts playing, all recording devices should start recording simultaneously. To implement this synchronization:
5. In the slot/method handling the **“Start Recording & Play Stimulus”** button:
   1. **Trigger phone recordings:** Invoke the function or signal that was implemented in earlier milestones to start recording on the smartphones. For example, if there is a method start\_all\_devices\_recording() in a RecordingController, call that. This might send a network message or an ADB command to each phone to begin recording (video or sensor data, depending on context). Ensure this call is non-blocking (if it waits for confirmation, it might delay things; ideally it should send the command asynchronously).
   2. **Start PC recording:** If the PC itself is recording (e.g., a webcam feed or screen capture), start that as well. For instance, if using OpenCV to capture the webcam, call the routine to begin capturing frames to file. If using a QCamera/QMediaRecorder for the webcam, trigger cameraRecorder.start() etc. This should also be as immediate as possible.
   3. **Play the stimulus video:** Immediately after starting the recordings, call self.mediaPlayer.play() to begin video playback. The order of these three actions can be adjusted slightly depending on which is more time-sensitive. If starting phone recordings takes a moment (network latency), you might call play video last. In practice, doing them back-to-back in code will result in near-simultaneous start (within tens of milliseconds). For better sync, you could introduce a very short delay to align start times, but this is likely unnecessary if we log the exact times (next step).
   4. **Record the start timestamp:** Capture the current system time at the moment of starting. You can use time.time() (which gives sub-second precision) or QDateTime.currentDateTime() for a formatted timestamp. Also, get the mediaPlayer’s position (which should be 0 or very close to 0 at start). Write an entry to the log, e.g.:
   * [2025-07-28 12:46:05.123] Experiment started, stimulus playback started (video=example.mp4)
   * Optionally include the file name and length of video if useful. Since all devices were triggered at this time, this timestamp serves as the **synchronization reference** for the experiment. Later, when analyzing data, one can align all recordings to this "t=0" reference.
6. It might be wise to disable the "Start Recording & Play" button after it’s clicked (to prevent accidental double trigger) or hide it while the experiment is running.
7. If the workflow requires it, you could enforce loading a video before enabling the start button.
8. **Checkpoint:** After implementing this, test the synchronized start without a second monitor (to keep it simple). Load a video, then click "Start Recording & Play Stimulus". Verify that:
   * The video starts playing.
   * You see indications that phone/PC recording started (depending on how previous milestones signaled that – perhaps a status message or LED on phones).
   * Check the log file after stopping: it should have a start entry with a timestamp. For now, you might just log to console or a text file. Example console output might be:
   * Experiment start: 1690538765.123 (Unix timestamp)
   * We will formalize the logging in the next step, but ensure you can capture the time. If possible, also verify that the phones indeed started recording at that command (perhaps by later checking their recorded file timestamps or durations to see if they match the video’s duration).
9. **Implement Logging of Stimulus Timing and Markers:** Logging is crucial for synchronization. We will maintain a log file (e.g., a simple CSV or text log) on the PC that captures key events and their times. Set up a logging mechanism as follows:
10. When the experiment starts (as above), open a log file in the project’s output directory (perhaps alongside where PC recordings are saved). The filename could include a timestamp or trial ID, e.g., experiment\_log\_20250728\_124605.txt. Write a header or initial line noting the start time.
11. **Stimulus Start:** Log the exact system time of stimulus playback start. Also note the video filename and perhaps the planned duration (you can get duration via mediaPlayer.duration() in milliseconds after the media is loaded).
12. **Marker events:** Connect the **“Mark Event”** button to a slot that logs a marker. When the operator clicks it (or if you decide to use a hotkey like "M"), retrieve the current playback position via mediaPlayer.position() (in milliseconds) and the current system time. Write a line to the log such as:

* [2025-07-28 12:46:15.842] Marker pressed – video time = 10.719 s
* This indicates that 10.719 seconds into the video (since play started), a marker was set. The system time is also recorded in brackets. Markers can be numbered or labeled if needed (for now, a generic "Marker pressed" is fine, or you can number them incrementally).

1. **Video end or stop:** It’s useful to log when the stimulus ends. QMediaPlayer emits a signal when playback is finished (stateChanged or mediaStatusChanged to EndOfMedia). You can connect a handler to log "Stimulus finished at T=... (duration = X)". If you plan to automatically stop recordings at the moment the video ends, this signal can trigger that (see next step).
2. For simplicity, use Python’s built-in file write operations for logging. Open the file in append mode and flush after each write to ensure data is not lost if the app crashes. Alternatively, use the logging module with a FileHandler.
3. Keep all logs **local on the PC** (requirement #3) – we do not send these timestamps to the phones. The phones are just capturing video, and their clocks might not be synced; the PC log is the master record for synchronization.
4. **Checkpoint:** Simulate an experiment run and check the log file. Ensure the times make sense. For example, if you put a marker roughly halfway, the video time in the log should correspond roughly to that portion of the video. If you have access to the phone videos after, you could visually confirm that at the logged times certain events coincide. (Full verification of sync might be complex, but at least ensure the logging is functioning.)
5. **Stop and Teardown Workflow:** Once the stimulus is done and enough data is collected, the operator will stop the recording:
6. If the video runs to completion, you could programmatically detect this and auto-stop the recordings. To do this, connect QMediaPlayer’s stateChanged signal. When the state changes to QMediaPlayer.StoppedState or when mediaStatusChanged indicates EndOfMedia, you can call the routine to stop recordings on all devices and close out the experiment. This ensures everything stops at the same time the stimulus ends. Auto-stopping is convenient, but make sure the operator is aware (so it doesn’t surprise them). You might print a message or enable a prompt, but likely it's fine to stop automatically at video end.
7. If the operator manually stops early (say they hit a "Stop" button to abort), then implement that: a Stop button could halt video playback (mediaPlayer.stop()) and send stop commands to devices. Log that the run was stopped manually.
8. In either case, once stopped, finalize the log (write a "Experiment stopped at ..." line and close the file).
9. Reset the UI state: re-enable buttons, allow loading a new stimulus for the next run, etc. Possibly also reset the media player (e.g., call mediaPlayer.stop() which should rewind to start, or reload the media if needed).
10. **Checkpoint:** Test stopping scenarios:
    * Let the video play to the end and see if everything stops (if you implemented auto-stop). Check that the log has an entry for stimulus end.
    * Restart another run to ensure the system can run multiple times sequentially (the media player and devices should be ready for a new session after stopping).
    * Test manual stop (if implemented): mid-video, click "Stop". The video should cease and devices stop recording. Check logs for a stop entry.
11. **Keyboard Shortcut Configuration:** We have touched on this earlier, but ensure these are working as required:
12. **Spacebar toggles Play/Pause:** Implemented either via QShortcut on the video widget or by reusing the Play button’s shortcut. For example, you could set the Play/Pause button’s shortcut to Space (but that might only work when the main window is focused). Better is attaching to the video widget as discussed. In code, the toggle action would be something like:

* def handle\_toggle\_play(self):  
   if self.mediaPlayer.state() == QMediaPlayer.PlayingState:  
   self.mediaPlayer.pause()  
   else:  
   self.mediaPlayer.play()
* This logic checks the current state and toggles[[9]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=def%20play%28self%29%3A%20if%20self,play). Connect this to Spacebar.

1. **Esc exits full-screen:** As described, either override keyPressEvent in the QVideoWidget subclass:

* def keyPressEvent(self, event):  
   if event.key() == Qt.Key\_Escape and self.isFullScreen():  
   self.setFullScreen(False)  
   event.accept()  
   else:  
   super().keyPressEvent(event)
* This will catch Esc. If you want to also pause or stop on Esc, you can add self.mediaPlayer.pause() or similar when exiting.

1. **Other keys:** If desired, you can add more shortcuts (like "F" for full-screen toggle, arrow keys for seeking, etc.), but those are optional and not requested. For now, confirm Space and Esc perform as expected.
2. **Checkpoint:** Final user interaction test:
   * While video is playing (windowed or full-screen), press Space. Video should pause; press again, it resumes.
   * While in full-screen, press Esc. It should exit full-screen (and possibly pause, if you implemented that).
   * If you press Esc when not in full-screen, ensure it doesn’t close the whole app by accident (Esc has no default action in a normal window, so it should be fine).
   * All these should work both when the QVideoWidget has focus (like in full-screen it will) and when the main window has focus (if you used ApplicationShortcut context for Space, it might work even when not focused on the button).
3. **Testing the Complete Flow:** Now put everything together to test a full experimental run simulation:
4. Launch the application, connect the devices (phones) as per earlier milestones.
5. **Step 1:** Load a sample video via "Load Stimulus". The video might start showing a first frame or be black (depending on Qt, often it doesn’t show a frame until play). No problem.
6. **Step 2:** Click "Start Recording & Play Stimulus". Immediately, verify:
   * Phones start recording (perhaps you have an indicator or you can hear a beep if you coded that, etc.).
   * PC webcam starts recording (maybe a light on the webcam turns on).
   * The video stimulus goes full-screen on the selected display and begins playing for the participant.
   * The log file is created with a start timestamp.
7. **Step 3:** During playback, click "Mark Event" once or twice at notable moments.
8. **Step 4:** Let the video play to completion. Ensure the video closes or exits full-screen (if you didn't automate this, press Esc manually when it’s done). Stop the recordings if not auto-stopped.
9. **Step 5:** Click "Stop" (if needed) to ensure everything is stopped.
10. After the run, inspect the outputs:
    * The phones should have video/sensor files of the duration roughly matching the video length.
    * The PC webcam video should also have that duration.
    * The log file should contain:
    * Start time,
    * Marker times,
    * End time. Check that the marker times (video position) make sense (e.g., if you marked at a specific scene, confirm in the video or in the recorded footage).
11. Address any discrepancies. For example, if the phone videos start a second later than the stimulus, perhaps there was a delay in sending the command; this would show up as the marker in phone video appears offset. Our logged times will help adjust for that in analysis. The key is all events are logged on the same timeline (PC’s clock), so alignment can be done post-hoc.

## Test Checkpoints and Verification

Throughout the implementation, we identified several checkpoints. Here is a summary of test checkpoints and what to verify at each stage:

* **Video Load & Play Test:** After integrating QMediaPlayer and QVideoWidget, verify that selecting a video file and clicking Play actually starts the video in the UI. The video should be visible and play smoothly, and you can pause/resume. This confirms the media player is functioning.
* **Full-Screen Display Test:** Ensure the stimulus can go full-screen (preferably on a second monitor). The entire screen should show the video with no window borders. Verify that pressing **Esc** returns from full-screen, and that **Space** can control playback while full-screen. If using a single monitor, test that Esc restores your app window properly after full-screen.
* **Synchronized Start Test:** When using the "Start Recording & Play Stimulus" button, confirm near-simultaneous start:
* The video should start and the recording devices should start together.
* Use system timestamps to confirm how close the actions are. For instance, log a timestamp before and after each action; in an ideal case they are within a few milliseconds. Minor delays are okay as long as we log accurately.
* Check that the log entry for start time is recorded.
* **Marker Logging Test:** During a test playback, press the "Mark Event" button (or corresponding hotkey) a few times. Later, open the log file:
* Ensure each press created a log entry with a timestamp and the video position (ms or seconds). The video position should correspond to the actual moment in the video (you can cross-check by the content of that moment).
* If possible, correlate with one of the recorded video streams (e.g., the phone’s video might show something at that marker time).
* **Complete Flow Test:** Do a full run as described (load -> start -> mark -> finish) and then examine *all* outputs:
* **Log file:** Has start, markers, end times.
* **Recorded files:** Are present and have correct duration.
* **No crashes or hangs:** The app should handle the sequence without freezing. The UI should stay responsive (you should be able to click "Mark Event" or stop).
* **Resource usage:** Observe CPU/memory during the run if possible. Playing a video and recording multiple streams is heavy; ensure the system can cope. If the video stutters or frame drops occur, consider using a lower resolution video or check if the disk writing (for recordings) is a bottleneck.
* **Edge Cases:** Test a few edge scenarios:
* Loading a very short video (a few seconds) and see if everything triggers and stops correctly.
* Loading a longer video but stopping early manually.
* Trying a video file that is not supported (the mediaPlayer might emit an error). Ensure your app doesn’t crash – you can connect mediaPlayer.errorOccurred signal to display a warning like "Failed to play video - unsupported format". This is optional but user-friendly.
* If no video is loaded and "Start Recording & Play" is pressed, handle gracefully (maybe disable the button until a video is loaded, or prompt "Please load a stimulus first").

Each checkpoint ensures that a part of the system works as intended. By the end of testing, you should have confidence that stimuli presentation is properly integrated and synchronized with data recording.

## Additional Considerations and Future Enhancements

* **Supported Stimulus Formats:** Currently, we decided to support **only video files** (common formats like MP4, AVI, etc.). Image slideshows or other media types are not implemented (as per requirements). In the future, if needed, support for image sequences (slideshow) could be added by preloading images and flipping through them with a timer. Also, support for audio stimuli or other modalities could be integrated via similar principles (Qt can play audio, or send signals to devices).
* **Accuracy of Synchronization:** Our method logs timestamps to synchronize data. For most purposes (especially with 30 or 60 FPS video and similar camera frame rates), logging start times is sufficient. If sub-frame accuracy is required, more sophisticated methods would be needed:
* We could embed a visual or audio sync signal in the stimulus (e.g., a flash or a beep at time 0) and detect that in the recordings.
* Or have the phone cameras in view of a screen that displays a timestamp counter.
* However, these are beyond the scope of software and lean into experiment design. Given typical use, our approach should be acceptable.
* **Using VLC or Alternative Players:** If QMediaPlayer doesn’t meet performance needs or codec support (for example, some high-bitrate or unusual format videos might not play well), integrating VLC could be considered. The python-vlc library can open a video in a separate window (or embed in a PyQt widget). It provides callbacks for timing as well. The trade-off is added complexity and an external dependency (the user would need VLC installed). For now, Qt’s native player is chosen for simplicity.
* **UI/UX Improvements:** To make the experiment run smoother, consider:
* Providing a **countdown or cue** before the stimulus starts (e.g., a "Ready... Set... Go!" or just a 3-2-1). This could help ensure the participant is attentive right at the start and that all systems are fully ready. This could be done by showing a blank screen or a countdown on the stimulus display, and then starting the video.
* Hiding the mouse cursor on the stimulus screen when full-screen (to avoid distraction).
* Locking controls during playback to prevent accidental interruptions (except the marker and stop).
* After the run, maybe automatically prepare the next run (reset state, etc., possibly part of the next milestones).
* **Marker Labeling:** Our marker button currently just logs a generic marker. In the future, you might allow the operator to label markers (e.g., "Stimulus A shown" or "Subject responded"). This could be done by having multiple marker buttons or a quick dialog that asks for a note when you press the marker. For now, the timing is captured, and details can be annotated later in analysis.
* **Data Integration:** Since all marker and stimulus times are logged on the PC, during post-processing one will have to align these with the phone videos and sensor data. Typically, one would use the start time as a reference. For example, if the phone video file has its own internal timestamp or you assume it started at t=0 of experiment, then a marker at 10.7s means go to 10.7s in that video to see what happened. We should ensure to keep the log file format clear (perhaps also log phone start times if we get a response from them). This is more about analysis than implementation, but important for the end use.

In conclusion, the Stimulus Presentation Controller adds a complex but powerful capability to our system. We carefully designed the UI and class structure to integrate video playback with multi-device recording. We chose PyQt5’s QMediaPlayer for seamless integration, implemented full-screen output for participant-facing stimuli, and ensured that all critical timing information is captured locally for synchronization. By following the above steps and verifying at each checkpoint, you will achieve a robust implementation for Milestone 3.5. This sets the stage for the next milestones, which might involve more sophisticated metadata logging, data management, or user interactions based on the stimuli presented.

[[1]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=class%20VideoPlayer%28QMainWindow%29%3A%20def%20__init__%28self%29%3A%20super%28%29,PyQt5%20Video%20Player) [[2]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=if%20fileName%20%21%3D%20%27%27%3A%20self,fromLocalFile%28fileName) [[3]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=layout%20%3D%20QVBoxLayout%28%29%20layout,playButton) [[4]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=self) [[5]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=fileName%2C%20_%20%3D%20QFileDialog.getOpenFileName%28self%2C%20,QDir.homePath) [[9]](https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/#:~:text=def%20play%28self%29%3A%20if%20self,play) PyQt5 Video Player with QMediaPlayer - CodersLegacy

<https://coderslegacy.com/python/pyqt5-video-player-with-qmediaplayer/>

[[6]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=The%20problem%20is%20that%20the,Qt%3A%3AApplicationShortcut) [[7]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=2) [[8]](https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel#:~:text=layout%20%3D%20QtWidgets,or) python - QVideoWidget in full-screen mode no longer responds to hotkeys or mouse wheel - Stack Overflow

<https://stackoverflow.com/questions/60442806/qvideowidget-in-full-screen-mode-no-longer-responds-to-hotkeys-or-mouse-wheel>