# Milestone 3.1: PyQt GUI Scaffolding and Application Framework

## Goals and Overview

Milestone 3.1 focuses on building the basic **GUI structure** of the controller application using PyQt. The goal is to create a scaffold of the main window and key interface panels **without yet implementing full device functionality**. Key objectives include:

* Designing a **Main Window layout** with a flexible two-column structure: a left panel for device status/controls, and a right panel for live video previews. The window will also include a top/bottom section for control buttons (e.g. Connect, Start, Stop, Calibration) and a status bar for messages.
* Implementing a **Device Status Panel** (left side) that will list connected devices (two phones in this context) with indicators for connection status (e.g. green for connected). This panel will update in real-time as devices connect or disconnect (simulated at this stage).
* Implementing a **Preview Area** (right side) for video feeds. Each device provides two video streams (RGB and thermal), so the UI will reserve space for these – for example, using tabs or frames for each device’s feeds. We will use QLabel widgets to display image frames via QPixmap (a simple way to show images in PyQt). Initially, this will just show placeholder images or text, with real video to be integrated later.
* Creating a **Stimulus Control Panel** for controlling visual stimuli playback. This panel will include UI elements like a video file chooser (to load a stimulus video), Play/Pause controls, a timeline slider, and an output screen selector (for choosing a secondary display to show the stimulus). These controls will be placed in the main window (likely at the bottom) and remain disabled or non-functional until later milestones.
* Adding a **Menu Bar and Status Bar**. The menu will have options such as “File → Exit”, “Tools → Settings” (for configuring device IPs or other settings in future), and “Help → About”. The status bar will display runtime messages (e.g. “Ready”, connection notifications, errors) at the bottom of the window[[1]](https://realpython.com/python-pyqt-layout/#:~:text=,very%20center%20of%20the%20window).
* Establishing a clean **application structure** in code: organizing the GUI, networking, and other logic into separate modules and classes. We will subclass QMainWindow for the main UI, and plan out additional modules (e.g. a network handler, calibration module, etc.) to keep the code organized. We’ll also outline how to use PyQt’s signals and slots for thread-safe communication between the GUI and background threads (for device I/O), and consider adding a simple logging mechanism (e.g. a log window or console output) for debugging messages.
* **Testing** the GUI on a development machine (Windows) to ensure that all components load correctly, the layout is responsive to resizing, and the application closes cleanly. No actual device connectivity or video streaming is implemented in this milestone – it’s purely the visual framework upon which functionality will be built.

By the end of this milestone, we will have a running PyQt application that shows the main window with all the placeholders for devices, video feeds, and controls, providing a foundation for upcoming development.

## Step 1: Environment Setup and Tools

Before coding the GUI, set up your development environment:

1. **Python & PyQt Installation** – Ensure you have a suitable Python 3.x version installed. Install PyQt (either PyQt5 or PyQt6, depending on project requirements; PyQt5 is common for broad compatibility). You can install via pip: for example, pip install PyQt5. This will include the essential Qt libraries for widgets, GUI controls, etc. Verify the installation by launching a Python shell and importing a PyQt module (e.g. from PyQt5 import QtWidgets).
2. **IDE Configuration** – Use an IDE or editor that is friendly to Python GUI development. **PyCharm** is a good choice (with its Python support and debugger), or **Visual Studio Code** with the Python extension. In your IDE, create a new project for the controller app. If using PyCharm, set up a virtual environment and install PyQt5 in that environment. In VS Code, select the correct Python interpreter (with PyQt installed) for the workspace.
3. **Qt Designer (Optional)** – PyQt allows designing interfaces visually using Qt Designer. You may install **Qt Designer** (on Windows, you can get it via pip install pyqt5-tools, which includes Designer) if you prefer to **design the UI graphically**. Designer lets you drag-and-drop widgets to create a .ui file, which can then be loaded or converted into Python code. This can speed up laying out complex dialogs or main windows. If you use Designer, ensure your IDE knows the path to the Designer executable for convenience. (Alternatively, you can design the UI entirely in code – we will illustrate doing it in code for clarity, but using Designer is equally valid.)
4. **Project Folder Structure** – Set up a base folder for your project (e.g. ControllerApp/). Within this, create subfolders for different parts of the application. For example, you might have a gui/ package for all GUI-related modules, a network/ package for networking and device communication code, and perhaps other packages like calibration/ for calibration logic, etc. This will keep code organized as the project grows. We will outline specific modules and classes in the next step.

With the environment ready and tools installed, you can proceed to create the application framework.

## Step 2: Project Structure and Class Design

Before writing code, it’s useful to plan the modules and classes we’ll need for the GUI scaffold. A clean separation will make the application easier to expand. Below is a proposed **class and module breakdown** for this milestone (and anticipating future milestones):

* **Main Application Entry** (main.py): This will be the startup script. It will create a QApplication instance, instantiate the Main Window, and start the PyQt event loop. (On Windows, ensure to protect the startup with if \_\_name\_\_ == "\_\_main\_\_": guard, to avoid issues if using multiprocessing.)
* **Main Window Class** (gui/main\_window.py): A subclass of QMainWindow that defines the primary UI. This class will set up the menu bar, toolbars (for control buttons), status bar, and the central widget layout which contains all main panels. The MainWindow will aggregate other UI components (device panel, preview panels, stimulus panel) either as separate widgets or directly in its layout. It will also handle high-level signals/slots (e.g., menu actions, button clicks).
* **DeviceStatusPanel** (gui/device\_panel.py): A QWidget or QFrame that lists the connected devices and their statuses. This could use a QListWidget or QTreeWidget internally. It will provide methods to add/remove devices and update their status (like setting an icon or text to indicate connection state). The MainWindow may contain an instance of this panel on the left side.
* **PreviewPanel/DeviceView** (gui/preview\_panel.py): A QWidget that holds the video preview for one device. Since each device has two feeds (RGB and thermal), this panel might contain two QLabel widgets (or more advanced video widgets later) arranged either vertically or horizontally. We might create one instance per device. Alternatively, we might not need a separate class initially and instead directly set up the layout for each tab in the MainWindow code – but separating it can make the code cleaner if it grows (for example, if each preview panel gets its own controls or processing).
* **StimulusControlPanel** (gui/stimulus\_panel.py): A QWidget for the stimulus controls at the bottom. It contains buttons (Load, Play, Pause), a slider, and a screen selector combo box. We can implement it as a standalone widget class for clarity, or build it inside MainWindow. A separate class is helpful if the logic for loading videos and controlling playback becomes complex.
* **Networking Module** (network/device\_client.py or similar): This module will handle communication with the phones (e.g., via sockets or other protocol). In this milestone, we won’t implement actual networking, but we plan for a DeviceClient class or a QThread that runs listening for device data. It will eventually emit signals when a device connects, disconnects, or when a new frame is received. For now, you might create a stub class or just a placeholder function that simulates device messages (for testing the UI).
* **Calibration Module** (calibration/calibration.py): Placeholder for calibration logic (e.g., capturing synchronized frames from devices for calibration, computing calibration parameters, etc.). Not used in this milestone, but we note its existence for future integration. The “Capture Calibration” button in the UI will trigger functions here in later milestones.
* **Logging/Utilities** (utils/logger.py or similar): We might set up a simple logging mechanism. At this stage, logging can simply be print statements or Python’s logging module output to console. If a GUI log panel is desired, we could create a QTextEdit in a dockable panel to show logs. For now, we may just plan this out.

Each class/module above will be fleshed out as needed. **Figure 1** below summarizes the core classes and their roles:

* *MainWindow*: Initializes and ties together all UI components; manages menu and status bar; owns instances of DeviceStatusPanel, etc.
* *DeviceStatusPanel*: Widget for device list (in MainWindow’s left panel). Provides UI for showing device connection status.
* *PreviewPanel* (per device): Contains two display widgets (for RGB and thermal video). In MainWindow’s right area (possibly in tabs).
* *StimulusControlPanel*: Widget for stimulus controls (bottom panel of MainWindow).
* *DeviceClient (Network thread)*: Runs in background (QThread), communicates with device; will emit signals like frameReceived or deviceConnected. (Not implemented fully now, but interface planned.)
* *Calibration logic*: Functions or class to handle calibration routines, invoked by UI actions (not implemented in this milestone).

This structured approach will make it easier to maintain the code. Now, let’s proceed with implementing the GUI step-by-step.

## Step 3: Creating the Main Window Skeleton

First, we create the **Main Window** using PyQt. This will be our main application window that holds all other components.

**3.1. Subclass QMainWindow**: In gui/main\_window.py, define a class MainWindow(QMainWindow). In its \_\_init\_\_, call the superclass constructor and then set up basic window properties: - Set the window title (e.g. "Device Controller" or any appropriate title). - (Optional) Set a default window size or geometry. For example, self.resize(1200, 800) to start with a decent size. Using QDesktopWidget or QScreen can retrieve screen size if needed to maximize or center the window.

**3.2. Central Widget and Layout**: QMainWindow requires a central widget to place content. We create a QWidget to serve as the central container:

central\_widget = QWidget(self)  
self.setCentralWidget(central\_widget)

Now decide on a layout for the central widget. Since we want a two-column layout (devices on left, previews on right), a horizontal layout makes sense. We can use QHBoxLayout:

main\_layout = QHBoxLayout(central\_widget)  
central\_widget.setLayout(main\_layout)

This layout (main\_layout) will be the top-level layout for our central widget. We will add two main child widgets to it (left panel and right panel). Using a bare central widget with a custom layout allows us to arrange multiple widgets freely, which is exactly what we need for a compound interface.

**3.3. Menu Bar**: Set up the menu bar at the top of the QMainWindow. PyQt’s QMainWindow comes with an empty menu bar by default. We can add menus like this:

menubar = self.menuBar()  
file\_menu = menubar.addMenu("File")  
tools\_menu = menubar.addMenu("Tools")  
help\_menu = menubar.addMenu("Help")

Add actions to these menus: - *File menu*: Create an “Exit” action. For example:

exit\_action = QAction("Exit", self)  
exit\_action.triggered.connect(self.close) # closes the window  
file\_menu.addAction(exit\_action)

- *Tools menu*: Add a “Settings” action. This might open a settings dialog in the future (for configuring device IPs, etc.). For now, it can just show a placeholder. E.g.:

settings\_action = QAction("Settings...", self)  
# Connect to a stub method that would open a settings dialog (not implemented yet)  
settings\_action.triggered.connect(self.show\_settings\_dialog)  
tools\_menu.addAction(settings\_action)

where show\_settings\_dialog could simply QMessageBox.information(self, "Settings", "Settings dialog not implemented yet.") as a placeholder. - *Help menu*: Add an “About” action. For example:

about\_action = QAction("About", self)  
about\_action.triggered.connect(self.show\_about)  
help\_menu.addAction(about\_action)

and define show\_about to display a QMessageBox with info about the application (name, version, author, etc.).

Now our main window has a functional menu bar with basic options (Exit actually closes the app, Settings and About show placeholders).

**3.4. Status Bar**: Initialize the status bar at the bottom of the window. QMainWindow provides self.statusBar() method which returns a QStatusBar object[[1]](https://realpython.com/python-pyqt-layout/#:~:text=,very%20center%20of%20the%20window). You can use it to show messages:

self.statusBar().showMessage("Ready")

This will display “Ready” in the status bar. Throughout the app, we will use the status bar to display short status updates (e.g., “Device 1 Connected” or “Calibration saved” messages). The status bar is automatically shown when you call self.statusBar() (in Qt Designer it’s created by default in a main window template).

At this point, if you run the application (instantiate MainWindow and call show() on it within a QApplication), you will see an empty window frame with the title, and an empty menu bar (with our menus) and a status bar showing "Ready". The central area will be blank because we haven’t added content yet. But we have the basic scaffolding: menu, status bar, and an empty central widget with a layout ready for our panels.

*Testing Checkpoint:* Run main.py to launch the app. Verify that: - The window comes up with the correct title. - The menu bar contains “File”, “Tools”, “Help” with the respective menu items. Click “About” and “Settings” to see the placeholder message, and “Exit” to ensure the app closes. - The status bar at bottom displays "Ready". - The central area is currently empty (we will fill it in subsequent steps), but no errors occur regarding layouts. The window should be resizable; try resizing and see that the (empty) central widget expands correctly (no fixed-size issues). This confirms the main window skeleton is ready.

## Step 4: Implementing the Device Status Panel (Left Column)

Now we fill in the left side of the main layout – the Device Status Panel. This panel will list the devices and show their status.

**4.1. Create the Panel Container**: We create a widget to hold the device list. It could be as simple as a QWidget with a vertical layout, or a QGroupBox with a title "Devices". For clarity, let's use a QGroupBox:

self.devicePanel = QGroupBox("Devices", self)  
device\_layout = QVBoxLayout(self.devicePanel)  
self.devicePanel.setLayout(device\_layout)

This gives a titled frame. We will populate device\_layout with the actual list widget and maybe other controls if needed.

**4.2. Device List Widget**: To display device entries, a few Qt widget choices are: - QListWidget: a simple list box of text items (we can make entries like "Device 1 – Connected"). We can also set an icon per item (e.g., a green/red circle for status). - QTreeWidget: allows a hierarchy or multiple columns (could be useful if we want to show additional info like battery level or IP address in future). But it’s a bit more complex if hierarchy isn’t needed. - QListView with a custom model: more flexible, but overkill for now.

For simplicity, use QListWidget. Create it and add to the layout:

self.deviceList = QListWidget()  
device\_layout.addWidget(self.deviceList)

Initially, this list is empty (no devices connected). We can, for testing, add placeholder items:

self.deviceList.addItem("Device 1 (Disconnected)")  
self.deviceList.addItem("Device 2 (Disconnected)")

Later, when the networking code detects a device, we will update this list (e.g., change "Disconnected" to "Connected" or highlight the item). We can also use item flags or roles to indicate status. For example, to simulate a connected device, we might set the item’s foreground color to green or use an icon. For now, just having text is fine.

**4.3. Status Indicator (Optional)**: To better indicate status, consider using icons. You can create QIcons from image files (like a green dot and red dot). If you have icon assets, load them and set via item.setIcon(QIcon("green.png")) for connected, etc. Alternatively, use Unicode colored emojis (🟢/🔴) in the item text as a quick hack for now. For instance: "Device 1 🟢" vs "Device 1 🔴". This is optional at this stage but helps visualize status. We’ll assume text is enough for now.

**4.4. Integration into Main Layout**: Add the devicePanel widget to the main layout (left side). Since main\_layout is an QHBoxLayout:

main\_layout.addWidget(self.devicePanel)

We may want the left panel to not stretch too wide when the window is large, so you can set a maximum or preferred width. For example:

self.devicePanel.setMaximumWidth(250)

to limit it to 250px. Or use main\_layout.setStretch(0, 0) and main\_layout.setStretch(1, 1) to give all extra space to the right side (index 0 = left widget, index 1 = right widget). Using stretch factors or a QSplitter (discussed below) ensures the preview area gets most of the room.

**4.5. (Optional) QSplitter Alternative**: Instead of a plain layout, you could use a QSplitter to separate the left and right panels. A QSplitter is a container that holds two (or more) widgets with a draggable divider, allowing the user to resize the panels. For example:

splitter = QSplitter(Qt.Horizontal)  
splitter.addWidget(self.devicePanel)  
splitter.addWidget(right\_panel\_container) # we'll create right\_panel in the next step  
main\_layout.addWidget(splitter)

By default, splitter gives equal space; you can adjust initial sizes via splitter.setSizes([250, 750]) etc. Using a splitter is not essential, but it's a nice UX improvement to let the user adjust the width of the device list vs preview area. You can implement it now or later. For the initial scaffold, a fixed layout with stretch is simpler.

At this point, the left panel should be functional.

*Testing Checkpoint:* Run the app again. Now the main window should show a left section titled "Devices" with two list entries (Device 1 and 2, marked disconnected). The rest of the window (to the right of the device panel) is still empty (we’ll fill it next). Verify that the left panel remains a reasonable size. Try resizing the window: the device panel should stay at its fixed or limited width (if set), and the rest of the space grows for the preview area. The device list should be scrollable if many items are added (not needed now, just check that the list widget is visible and sized properly). No functional behavior yet – just the UI elements.

## Step 5: Implementing the Preview Area (Right Column)

Next, we build the right side of the interface – the preview panels for live video feeds from each device. We have two devices, each providing an RGB camera feed and a thermal camera feed. We’ll design this area to accommodate both devices’ feeds.

**5.1. Tabbed Interface vs Combined View**: As mentioned in the plan, we have options: - A **tabbed view**: one tab per device. The user can switch between “Device 1” and “Device 2” tabs to see that device’s feeds. - A **combined view**: show all feeds at once (e.g. a 2x2 grid for two devices × two feeds). This might be cluttered and shrink the videos, so the tabbed approach is cleaner initially.

We’ll implement a tabbed view for now (one tab per device). This is easily done with QTabWidget.

**5.2. Create the Tab Widget**:

self.previewTabs = QTabWidget()

We will create two tabs on this widget. For each tab, we need a widget that contains the two feed displays.

**5.3. Designing a Device Preview Panel**: For each device, we want to display an RGB image and a thermal image side by side or one above the other. A vertical stack might be better if the images are wide (so each gets full width), whereas side-by-side might be good if the images are tall. Suppose the feeds are likely in landscape orientation; stacking vertically might make them too small vertically. Side-by-side might be okay if window is wide. We can also place them in a grid (2x1 or 1x2). Here, we choose a vertical stack for simplicity (one on top of the other) so that each feed gets as much horizontal space as possible. We can always adjust later.

For **Device 1** tab: - Create a QWidget container: device1\_widget = QWidget(). - Set a vertical layout: vbox = QVBoxLayout(device1\_widget). - Create two QLabel placeholders:

rgb\_label1 = QLabel("RGB Camera Feed")  
thermal\_label1 = QLabel("Thermal Camera Feed")

Initially, just set text so we can see them. We will later replace this text with actual video frames. We might also give them a fixed size or minimum size for testing. For example:

rgb\_label1.setMinimumSize(320, 240)  
rgb\_label1.setStyleSheet("background-color: black; color: white;")

This will make the label visible (black background simulating a screen, white text). Do similarly for thermal\_label1. The styles are just to distinguish the area – you could also use a border: e.g. setStyleSheet("border: 1px solid gray;"). - Add these labels to the layout:

vbox.addWidget(rgb\_label1)  
vbox.addWidget(thermal\_label1)  
vbox.addStretch(1) # add stretch to push them to top, if we want some flex space

(The stretch is optional – it just ensures if the tab is larger than the total of labels, extra space goes below, keeping labels top-aligned. Without it, the labels will stretch/squash with the tab.) - Now add the tab to the QTabWidget:

self.previewTabs.addTab(device1\_widget, "Device 1")

Repeat for **Device 2**: - Create device2\_widget similarly with its own labels (rgb\_label2, thermal\_label2). - Possibly, to simulate two different feeds, label them distinctly or use different placeholder colors.

rgb\_label2 = QLabel("RGB Camera Feed")  
rgb\_label2.setStyleSheet("background-color: black; color: white;")  
thermal\_label2 = QLabel("Thermal Camera Feed")  
thermal\_label2.setStyleSheet("background-color: black; color: white;")

- Add them to a layout and then to a second tab:

self.previewTabs.addTab(device2\_widget, "Device 2")

Now self.previewTabs holds two tabs, each containing two labels.

**5.4. Add Preview Tabs to Main Layout**: We add the tab widget to the main layout (to the right side):

main\_layout.addWidget(self.previewTabs, stretch=1)

Using stretch=1 (and no stretch on device panel or stretch=0) will ensure the tab widget expands to fill remaining space. If using QSplitter, you'd have added the tab widget to the splitter earlier.

**5.5. Image Display Strategy**: At this point, the preview area is set up with placeholders. In the future, when actual video frames come from the devices, we will need to display them in these QLabel widgets. The typical approach is: - Convert incoming frames (e.g., from OpenCV or raw bytes) to a QImage, then to QPixmap, and set it on the QLabel using label.setPixmap(pixmap). QLabel automatically updates to show the pixmap. This approach is straightforward. - Ensure this update happens in the GUI thread. If frames are received in a background thread, we emit a signal carrying the frame, and connect that signal to a slot (method) in the MainWindow that calls setPixmap. PyQt’s signal-slot mechanism will handle thread-safety by queuing the call to the main thread.

For completeness, here’s how we might convert an image (for example, a NumPy array from OpenCV in BGR format) to QPixmap in the future:

def convert\_frame\_to\_pixmap(frame: np.ndarray) -> QPixmap:  
 rgb\_image = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB) # convert BGR to RGB  
 h, w, ch = rgb\_image.shape  
 bytes\_per\_line = ch \* w  
 qimage = QImage(rgb\_image.data, w, h, bytes\_per\_line, QImage.Format\_RGB888)  
 qimage = qimage.scaled(desired\_width, desired\_height, Qt.KeepAspectRatio)  
 return QPixmap.fromImage(qimage)

This is a common snippet to go from OpenCV image to QPixmap. In the actual app, the desired\_width/height would be the size of the QLabel or some standard size. Then we would do rgb\_label1.setPixmap(pixmap) to show it. For now, we are not implementing this – but it’s important to know the plan for later.

**5.6. (Optional) PC Webcam Preview**: The spec mentioned possibly showing the PC’s own webcam feed as well. If needed, we could add another tab (e.g. “PC Camera”) and display the PC’s webcam feed similarly. This might be done using OpenCV or QCamera from Qt. Since it's not confirmed as a requirement yet, we can leave it for later. Our structure can easily accommodate an extra tab if needed.

*Testing Checkpoint:* Run the app with the new preview area. Now the window should show: left side device list, right side a tab widget with “Device 1” and “Device 2” tabs. Each tab contains two dark rectangles (our labels) labeled “RGB Camera Feed” and “Thermal Camera Feed”. Test switching tabs – it should work (PyQt’s QTabWidget handles this automatically on click). Try resizing the window larger: the labels should expand (since we didn’t fix their size beyond a minimum). They likely stretch vertically to fill the tab. Try making the window smaller: the labels will shrink accordingly (down to their minimum size). The layout should remain sensible (labels not overlapping, text still visible). Everything is still static, but the UI elements are in place.

## Step 6: Adding Top Control Buttons (Toolbar)

The main window needs control buttons for actions like Connect/Disconnect to devices, Start/Stop a session, and capturing calibration frames. We’ll add these as a toolbar or a horizontal button bar.

**6.1. Choose Toolbar vs. Widget**: Since QMainWindow supports toolbars, we can use a QToolBar to hold these buttons. A toolbar can have QAction items with icons or text, and can be docked at top, bottom, or sides. Alternatively, we could place a row of QPushButton at the top of the central widget. Using a QToolBar is more standard for an application’s main control buttons, so we’ll do that.

**6.2. Create the Toolbar**: In MainWindow’s init (after setting up menu), do:

toolbar = self.addToolBar("MainControls")  
toolbar.setMovable(False) # lock it in place (optional)

Now create actions for each control: - Connect to Devices:

connect\_action = QAction("Connect", self)  
toolbar.addAction(connect\_action)

Similarly a Disconnect action:

disconnect\_action = QAction("Disconnect", self)  
toolbar.addAction(disconnect\_action)

We might later change these to a single toggle or disable one when the other is active, but for simplicity we include both. - Start Session:

start\_action = QAction("Start Session", self)  
toolbar.addAction(start\_action)

- Stop Session:

stop\_action = QAction("Stop", self)  
toolbar.addAction(stop\_action)

- Capture Calibration:

calib\_action = QAction("Capture Calibration", self)  
toolbar.addAction(calib\_action)

You can also add separators for grouping:

toolbar.addSeparator()

for example, to separate Connect/Disconnect from Start/Stop if desired.

We now have five actions on the toolbar (text-only for now). If you have icon images (like play/stop icons or connect icons), you can do QAction(QIcon("icon.png"), "Text", self) to make them prettier. In absence of icons, text is fine.

**6.3. Connect Actions to Slots**: These actions currently do nothing when clicked. We should connect them to methods, even if just placeholders:

connect\_action.triggered.connect(self.handle\_connect)  
disconnect\_action.triggered.connect(self.handle\_disconnect)  
start\_action.triggered.connect(self.handle\_start)  
stop\_action.triggered.connect(self.handle\_stop)  
calib\_action.triggered.connect(self.handle\_capture\_calibration)

Then implement these methods in MainWindow. At this stage, they might simply log messages or update UI in a trivial way:

def handle\_connect(self):  
 # Placeholder: In future, initiate connection to devices  
 self.statusBar().showMessage("Connect pressed - (simulation) connecting devices...")  
 # For now, simulate immediate connection:  
 for i in range(self.deviceList.count()):  
 item = self.deviceList.item(i)  
 item.setText(f"Device {i+1} (Connected)")  
 # maybe also switch icon to green, etc.

The above simulates that pressing Connect will mark all devices as connected. Similarly:

def handle\_disconnect(self):  
 self.statusBar().showMessage("Disconnect pressed - (simulation) disconnecting devices...")  
 for i in range(self.deviceList.count()):  
 item = self.deviceList.item(i)  
 item.setText(f"Device {i+1} (Disconnected)")  
}  
def handle\_start(self):  
 self.statusBar().showMessage("Session started (simulation)")  
def handle\_stop(self):  
 self.statusBar().showMessage("Session stopped (simulation)")  
def handle\_capture\_calibration(self):  
 self.statusBar().showMessage("Capturing calibration (simulation)")  
 # Here we would trigger calibration capture in future.  
}

These simply update the status bar with a message. We also simulate connect/disconnect by toggling the text in the device list. This gives some interactivity to test the UI. In a real scenario, handle\_connect would start network threads or send commands to devices; handle\_start might begin recording or data streaming; etc., but that’s for later milestones.

**6.4. Toolbar Placement**: By default, addToolBar() puts it at the top of the window (just below the menu bar)[[1]](https://realpython.com/python-pyqt-layout/#:~:text=,very%20center%20of%20the%20window). If we wanted it at the bottom, we could do self.addToolBar(Qt.BottomToolBarArea, toolbar). The spec said “along the top or bottom, include control buttons” – we’ve chosen top by using the default.

*Testing Checkpoint:* Run the app. Now you should see a toolbar with buttons: “Connect | Disconnect | Start Session | Stop | Capture Calibration” at the top. Click each: - Connect: should change device list items to “(Connected)” and status bar message “Connect pressed…”. - Disconnect: should change them back to “(Disconnected)” and status message. - Start/Stop/Calibration: just update the status bar message. Ensure the toolbar is not overlapping the central widget – it should be in its own area above the status bar and above the central content. The rest of the UI (device list and preview tabs) should appear below the toolbar. The layout should still be correct. This confirms our control buttons are integrated. (Later, we might disable some buttons contextually, e.g. disable “Start” until devices are connected, etc., but for now no such logic).

## Step 7: Adding the Stimulus Control Panel (Bottom Panel)

Now we tackle the **Stimulus Control Panel**, which contains controls for loading and playing a stimulus (like a video to display to the participant on a separate screen). According to the plan, it includes a file chooser, play/pause, a timeline slider, and output screen selection. We will add this UI at the bottom of the main window.

**7.1. Panel Container**: We can reuse approaches similar to device panel. For example, a QGroupBox titled "Stimulus". Alternatively, a simple QFrame or QWidget with a layout is fine (title can be given via a QLabel inside it, since we might not want a frame visible). Let’s use QGroupBox for consistency:

self.stimulusPanel = QGroupBox("Stimulus Controls", self)  
stim\_layout = QHBoxLayout(self.stimulusPanel)  
self.stimulusPanel.setLayout(stim\_layout)

We’ll use a horizontal layout to place controls in one row. If it becomes too many elements for one row, we might switch to a two-row layout, but let’s attempt a single row first.

**7.2. File Selection Control**: To load a stimulus file (likely a video file), we provide a way to choose a file. Common UI pattern: - A QLineEdit to display the selected file path (or name). - A “Browse” button (QPushButton) to open a file dialog.

Add these:

self.stimFilePath = QLineEdit()  
self.stimFilePath.setPlaceholderText("No file loaded")  
self.stimFilePath.setReadOnly(True)  
browse\_btn = QPushButton("Load Stimulus…")

We mark the QLineEdit read-only, because we want users to use the dialog, not manually edit the path (to avoid invalid paths). The placeholder shows if no file is loaded.

Now connect the browse button:

browse\_btn.clicked.connect(self.browse\_stimulus\_file)

And implement browse\_stimulus\_file to open a file dialog:

def browse\_stimulus\_file(self):  
 fname, \_ = QFileDialog.getOpenFileName(self, "Select Stimulus Video", "", "Video Files (\*.mp4 \*.avi);;All Files (\*)")  
 if fname:  
 self.stimFilePath.setText(fname)  
 self.statusBar().showMessage(f"Loaded stimulus: {os.path.basename(fname)}")  
 # In future, we would also load the video into a playback mechanism.

This will allow the user to choose a video file and display its path. (We need from PyQt5.QtWidgets import QFileDialog at top.)

Add the line edit and button to the layout:

stim\_layout.addWidget(self.stimFilePath)  
stim\_layout.addWidget(browse\_btn)

**7.3. Play/Pause Buttons**: Add buttons to play and pause the stimulus:

play\_btn = QPushButton("Play")  
pause\_btn = QPushButton("Pause")  
stim\_layout.addWidget(play\_btn)  
stim\_layout.addWidget(pause\_btn)

We will not implement actual playing now, but we can simulate:

play\_btn.clicked.connect(lambda: self.statusBar().showMessage("Play stimulus (simulation)"))  
pause\_btn.clicked.connect(lambda: self.statusBar().showMessage("Pause stimulus (simulation)"))

Optionally, disable these buttons until a file is loaded:

play\_btn.setEnabled(False)  
pause\_btn.setEnabled(False)

And in browse\_stimulus\_file, after setting the text:

if fname:  
 play\_btn.setEnabled(True); pause\_btn.setEnabled(True)

So they enable when a video is selected. This is a minor UX detail.

**7.4. Timeline Slider**: Add a QSlider for the timeline. Use a horizontal slider:

timeline\_slider = QSlider(Qt.Horizontal)  
timeline\_slider.setRange(0, 100) # 0 to 100% as a dummy range  
timeline\_slider.setValue(0)  
stim\_layout.addWidget(timeline\_slider)

This slider represents video progress. If we load a real video, we might set its range to the number of frames or the duration in seconds. For now, 0-100 is fine. We might also want to show the current time and duration, but skip that for now.

If video was playing, we’d move this slider accordingly. For now, we could connect it to a dummy:

timeline\_slider.sliderMoved.connect(lambda val: self.statusBar().showMessage(f"Seek to {val}% (simulation)"))

Just to see feedback when user drags it.

**7.5. Output Screen Selector**: If the experiment uses a dual-monitor setup (one monitor for operator UI, another to display stimuli full-screen), we need to choose which screen to show the stimulus on. A QComboBox can list available screens. PyQt can get screens via QApplication.screens():

screen\_combo = QComboBox()  
screens = QApplication.screens()  
for i, screen in enumerate(screens):  
 screen\_name = screen.name() or f"Screen {i+1}"  
 screen\_combo.addItem(f"{screen\_name} ({screen.size().width()}x{screen.size().height()})")

This will populate the combo with entries like “Screen 1 (1920x1080)” etc. If only one screen is present, it will just show that. We might default to second screen if available:

if len(screens) > 1:  
 screen\_combo.setCurrentIndex(1) # select second screen by default

Add the combo to layout, perhaps with a label:

screen\_label = QLabel("Output Screen:")  
stim\_layout.addWidget(screen\_label)  
stim\_layout.addWidget(screen\_combo)

We could disable the screen combo until a file is loaded or until a session starts, but probably it’s fine enabled (just selection doesn’t do anything yet). In the future, this selection would be used to position the stimulus window on the chosen screen.

**7.6. Finalizing Layout Placement**: Now we have added all widgets to stim\_layout. We need to place stimulusPanel in the main window. Since in our main layout (which is currently an HBox for left/right) we didn’t yet account for a bottom row, we have a couple of ways: - Change the central widget’s layout to a QVBoxLayout so we can have two rows: top = HBox (device+preview), bottom = stimulusPanel. - Use QMainWindow’s dock area: place stimulusPanel as a bottom dock widget.

The simpler approach is to adjust the central layout. We can nest our existing HBox into a VBox. One way: - Create a vertical layout for central widget instead of horizontal at the start. - Put the HBox (devicePanel + previewTabs) into a container widget or layout item inside that vertical layout. - Then add the stimulusPanel below it.

If we planned ahead, we could have done:

central\_vlayout = QVBoxLayout(central\_widget)  
top\_panel = QWidget()   
top\_hlayout = QHBoxLayout(top\_panel)  
# ... add devicePanel and previewTabs to top\_hlayout ...  
central\_vlayout.addWidget(top\_panel)  
central\_vlayout.addWidget(self.stimulusPanel)

Alternatively, since we already had main\_layout = QHBoxLayout(central\_widget), we can cheat by adding the stimulus panel as another widget to the main\_layout – but that would put it to the right of previewTabs (which is wrong). So, we do need the vertical layout.

**Refactoring to Vertical Layout**: - Instead of main\_layout = QHBoxLayout(central\_widget), do:

central\_vlayout = QVBoxLayout(central\_widget)  
top\_panel = QWidget()  
top\_hlayout = QHBoxLayout(top\_panel)  
top\_panel.setLayout(top\_hlayout)  
central\_vlayout.addWidget(top\_panel)

- Then where we did main\_layout.addWidget(self.devicePanel) and main\_layout.addWidget(self.previewTabs), we now do:

top\_hlayout.addWidget(self.devicePanel)  
top\_hlayout.addWidget(self.previewTabs, 1)

(the ,1 stretch is to give the preview more space as before). - Finally, add the stimulus panel:

central\_vlayout.addWidget(self.stimulusPanel)

This way, the device+preview occupy the top part, and stimulusPanel is directly below. By default, the vertical layout will allocate space based on content’s size hints. Our stimulusPanel is relatively small (horizontally laid out controls), so it will take minimal height, and the top\_hlayout will take the rest. If necessary, we can enforce relative sizes: e.g., central\_vlayout.setStretch(0, 1) (for top panel) and central\_vlayout.setStretch(1, 0) for bottom panel to give top panel all extra space.

Make sure to remove or adjust any previously set stretch on main\_layout because we replaced it. If using QSplitter earlier, we can still embed splitter inside top\_panel instead of manual HBox.

This refactor is a bit of work, but conceptually straightforward. If you prefer not to refactor, an alternative hack: place stimulusPanel in the status bar. Qt allows adding widgets to the status bar (on the right side typically). But the stimulus controls are too many for a status bar and it’s semantically different, so better to do the proper layout.

After adjusting to vertical layout, the MainWindow central widget now contains two sections stacked: - The upper section: a horizontal split of device list and preview tabs. - The lower section: the stimulus controls panel.

**7.7. Adjusting Appearance**: Perhaps give the stimulus panel a slight margin or border to distinguish it. A QGroupBox by default draws a frame and title, which is okay. The title "Stimulus Controls" will show. If you prefer no title but a cleaner look, you could use QFrame with a styled border or just a layout. We’ll stick with the titled group box as it clearly labels the section.

*Testing Checkpoint:* Run the app with the new layout. The window should now show: - Top left: devices list, top right: tabbed previews (same as before). - Bottom: a group box "Stimulus Controls" containing a read-only text field (initially "No file loaded"), a "Load Stimulus…" button, Play, Pause, a slider, and "Output Screen" combo. These should all line up in one row. If the window is not wide enough, they might compress or the layout might start to squeeze the text field. Stretch the window wide to see all controls. The slider should stretch to take up any extra space in that row, ideally. If it’s too short, we might need to set timeline\_slider.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Fixed) to make it grow. Ensure that when no file is loaded, Play/Pause are disabled (if you implemented that). Click "Load Stimulus…" – a file dialog should appear. Select a video or any file; the path appears in the text field, and Play/Pause enable. (We’re not actually playing the video, but the UI responds.) Move the slider – it should move and the status bar will show the value (thanks to our lambda). Change the "Output Screen" combo if you have multiple monitors; nothing visible happens, but we have the value.

Also verify that the resizing behavior is good: The bottom panel should stay at a fixed height (determined by the size of its widgets) and the top panel (device+preview) should expand when you resize taller. If the bottom panel is taking too much space or too little, adjust the stretches. For width, the bottom panel spans across the whole window width (which is fine). The presence of the bottom panel shouldn’t disturb the top layout beyond reducing its vertical space. Everything should still be nicely contained within the main window.

Now our GUI scaffold is fully laid out with all sections.

## Step 8: Integrating Backend Signals and Logging (Planned)

With the GUI components in place, we should consider how the **application logic** will hook into this UI. This involves using threads for device communication, PyQt signals/slots for updating the GUI, and possibly a logging system for debug output. In this milestone, we won’t implement the full backend, but we prepare the framework.

**8.1. Device Communication via Threads**: The two phones will likely send video frames and status info over the network. We must not block the GUI while waiting for this data. The typical solution is to run the network I/O in separate threads (e.g., subclass QThread or use QThreadPool with QRunnables). Those threads, upon receiving data, will emit signals to the main thread. For example, a DeviceClient thread could have:

class DeviceClient(QThread):  
 frameReceived = pyqtSignal(int, np.ndarray) # device index and frame data  
 statusChanged = pyqtSignal(int, bool) # device index and connected/disconnected  
 ...

When it gets a frame, it does self.frameReceived.emit(device\_index, frame). In the main GUI (MainWindow), we connect these signals:

device\_client.frameReceived.connect(self.update\_frame)  
device\_client.statusChanged.connect(self.update\_device\_status)

And implement update\_frame(device\_index, frame) to convert the frame to QPixmap and set it on the appropriate QLabel (device 1 RGB or thermal, etc.). This way, the GUI update happens in the GUI thread, ensuring thread safety (Qt’s signals/slots handle the thread crossing safely). We plan for such methods, though we won’t see them in action until the backend is built. We can, however, **simulate** a frame update with a QTimer or a dummy thread that emits signals after a delay to test if our labels update correctly (if we implement a stub of update\_frame that just changes the label text or color).

**8.2. Updating Device Status**: Similarly, when a device connects, the network thread would emit statusChanged(device\_index, True). Our slot update\_device\_status would then update the device list item – e.g., set the text to "Connected" and maybe color green. We already simulate this in handle\_connect/handle\_disconnect. Later, we will replace that simulation with real signal handling.

By planning these signals and slots now, we ensure our UI elements (like deviceList and the labels) are accessible/updatable from those slots. For example, keep references like self.rgb\_label1 as members if needed, so update\_frame can do self.rgb\_label1.setPixmap(new\_pixmap).

**8.3. Logging Mechanism**: As the app grows, having logs is helpful for debugging (and possibly for the user to see what’s happening). We have a status bar for short messages, but a multiline log could record detailed events (device IP, errors, etc.). One approach: - Use Python’s logging module to log to console or file. - Additionally, create a QTextEdit or QListWidget in the UI to show logs. We could put this in a QDockWidget so it can be shown/hidden. For example, a dock titled "Log" that appears at the bottom (or side) with a text box.

If we want to set this up now:

log\_dock = QDockWidget("Log", self)  
log\_widget = QTextEdit()  
log\_widget.setReadOnly(True)  
log\_dock.setWidget(log\_widget)  
self.addDockWidget(Qt.BottomDockWidgetArea, log\_dock)

This adds a dockable panel. We can append log messages to log\_widget via log\_widget.append("Device 1 connected.") etc. For now, we might not add every message, but hooking our existing actions to log could be useful. For example, in handle\_connect, after updating UI, also do:

log\_widget.append("Connect button pressed: simulation of connecting devices.")

This way, the user/developer can see a running log. If implementing, ensure to import QDockWidget, QTextEdit from QtWidgets.

However, adding a log window is optional in this milestone. It was suggested (“writing to a QTextEdit log panel or console output for debug messages” in the plan). If time permits, implement it, or else rely on console prints. Since we already have status bar messages, logs might be redundant at this point.

**8.4. Summary of Connections**: After this milestone, the GUI is structured but mostly static. To summarize how things will connect in the future: - **Connect/Disconnect buttons**: will trigger starting or stopping the DeviceClient threads which manage actual connections. The UI currently just simulates immediate connection; later it will initiate a connection process and maybe disable the Connect button while connecting. - **Start/Stop Session**: will control recording or data collection. Possibly signals to devices to start sending data, and enable the video feed display. - **Capture Calibration**: will instruct the devices (or use their last frames) to capture a calibration snapshot. The calibration module might process these frames (e.g., find a checkerboard, etc.). The UI might show a message or highlight that calibration was done. - **Stimulus controls**: when Play is pressed, we will open the chosen video (using a media player library or OpenCV) and display it on a second screen fullscreen. The slider will move as video plays, and Pause will pause the playback. Implementing this will involve either QtMultimedia (QMediaPlayer) or an external video player approach. We might decide to handle it in a separate thread or process to ensure smooth playback. - **Device video feeds**: The labels will be updated in real-time via signals as described. We might also need to scale the video frames to fit the label size (we did use Qt.KeepAspectRatio in our conversion example to maintain aspect ratio). If performance becomes an issue, we could consider using a more advanced widget like QGraphicsView or a QVideoWidget (from QtMultimedia) for video, but QLabel is often sufficient for moderate frame rates.

At this stage, **no backend code is fully running**, but our GUI is ready to integrate with it. We have considered the design so that adding those pieces will be straightforward.

## Step 9: Testing the GUI Scaffold

Finally, conduct a thorough test of the GUI application as it stands. Here are the test checkpoints and expected outcomes:

* **Launch Test**: Run the application (python main.py). The main window should appear without errors. Verify window title and that all major sections (menu, toolbar, device panel, preview tabs, stimulus panel, status bar) are visible.
* **Layout and Resizing**: Try resizing the window to various sizes:
* Make it large: the preview video labels should expand, the device list stays at a set width (or expands a bit if no max width set, but should not vanish), the stimulus panel stays at bottom with its components spread out nicely.
* Make it small: the layout should shrink proportionally. Check that the device list doesn’t disappear (it might get a scrollbar if too narrow). The tab widget might compress the labels – that’s fine as a stress test. The stimulus panel might start to compress its elements (text field might truncate text, etc., which is acceptable to a degree). The window has a minimum size determined by these widgets; ensure it doesn’t become unusably small (you can set setMinimumSize on some widgets if needed).
* The splitter (if used) should allow dragging divider and the layout should update accordingly. The vertical proportion of top vs bottom panel is fixed by layout stretch; you can’t drag that (unless we used another splitter vertically). But the bottom panel has a fixed minimal height so it should be okay.
* **Menu Actions**: Click each menu item:
* File → Exit should close the app immediately.
* Tools → Settings should show the placeholder dialog or message ("not implemented" message box). Close that and continue.
* Help → About should show the about information.
* No menu action should cause a crash or freeze.
* **Toolbar Buttons**: Click the toolbar buttons in various sequences:
* Connect: Status bar should update message, device list text changes to “(Connected)”. If you click again (Connect) repeatedly, it will just repeat the message and maybe reset text (idempotent in our simulation).
* Start Session: status bar shows "Session started". You could click Start even if not connected in our simulation; it still works (in a real app, we might disable it until connected).
* Stop: shows "Session stopped".
* Capture Calibration: shows calibration message.
* Disconnect: changes device list back to “(Disconnected)”. Try pressing Start after disconnect, etc., just to see nothing breaks. Our stub methods don’t have dependencies, so it should be fine.
* **Device List Interaction**: Optionally, test selecting items in the device list. By default, QListWidget allows selection. It doesn’t do anything yet, but we could later use selection to indicate which device’s tab to show or show device info. For now, just ensure clicking on an item doesn’t cause any issue. Also, if you double-click, nothing special happens (unless default is edit text, which we can disable via setEditTriggers(QAbstractItemView.NoEditTriggers) if needed).
* **Tab Switching**: Switch between Device 1 and Device 2 tabs. Make sure both have their labels. There’s no dynamic content, so this is just a UI toggle test.
* **Stimulus Controls**:
* Click “Load Stimulus…”: File dialog appears. Cancel it – nothing should break (no path set). Open it again, this time select a file (preferably a video file to follow the filter, but any file works as test). After selection, the QLineEdit should show the file path, and Play/Pause should enable. The status bar shows a loaded message with filename. If you select a very long path, ensure the QLineEdit is long enough (it will scroll horizontally if needed).
* Click Play: status bar “Play stimulus (simulation)”. Click Pause: “Pause stimulus”. You can click Play multiple times; it just repeats the message.
* Move the timeline slider: status bar should update with e.g. “Seek to 37%”. Try dragging it around.
* Change the Output Screen combo (if you have more than one screen, try selecting different entries). Currently, we didn’t connect it to anything, but just changing selection should do nothing (maybe we could have a signal to status bar too, but it’s okay).
* **Closing the App**: Test exiting via different means:
* Menu File → Exit.
* Clicking the window’s close [X] button.
* If you started via console, hitting Ctrl+C (not typical for GUI, but just in case). The application should terminate cleanly. Because we haven’t started any threads or timers (except maybe the slider events), there should be no hanging background process. If you did use a QTimer or something for simulation, ensure to stop it on close (not needed in our current setup).

Everything in the GUI is purely in the main thread for now, so we expect a smooth close. In later versions, we must ensure to stop threads on exit (as the code snippet from the video example noted, releasing camera or stopping threads in closeEvent).

## Step 10: Conclusion and Next Steps

Congratulations – we have built a comprehensive PyQt GUI scaffold for the controller application. At this point, the application provides a **framework of interactive UI components**: a device list, video feed panels, control buttons, and stimulus controls, all integrated in a cohesive main window. The design follows Qt best practices (using QMainWindow with menu, toolbars, status bar, and a central widget layout)[[1]](https://realpython.com/python-pyqt-layout/#:~:text=,very%20center%20of%20the%20window). The UI is organized and ready to be connected to real functionality.

In upcoming milestones, we will proceed to **implement the backend logic** and tie it into this GUI: - Establish actual network connections with the device phones. Likely, each phone will run a client that our app connects to or vice versa. We’ll spin up threads to handle communication and emit signals when data is received. The GUI will react by updating the Device Status Panel (e.g. turning indicators green when connected, showing battery status if available, etc.) and feeding video frames into the Preview Area labels in real-time. The use of PyQt signals ensures these updates are thread-safe. - Implement the **video display** properly: converting incoming frames to QPixmap and scaling them to fit the QLabel. We’ll reuse the approach discussed (cv2 to QImage to QPixmap) for the RGB and thermal streams. Performance considerations (like possibly downscaling if the GUI can’t handle full resolution at high FPS) will be addressed. - Add functionality to the **Start/Stop Session** buttons: possibly controlling recording of data or synchronization of the two device streams. - Implement **Calibration capture**: when triggered, instruct devices to capture a frame (or use the last frame) and run calibration routines (maybe saving images, computing alignment between RGB and thermal, etc.). The UI might give feedback like “Calibration successful” or display calibration data. - Flesh out the **Stimulus Player**: integrate a way to display the chosen video (or image sequence) full-screen on the selected output screen. We might use Qt’s QMediaPlayer and a QVideoWidget for ease, or a custom approach with OpenCV if we need frame-by-frame control. The Play/Pause buttons will control the playback, and the slider will reflect and control the playback position. - Add any additional polish to the UI: for example, showing battery levels (maybe as a progress bar or icon in the device list), adding an “elapsed time” label for session, improving the styling (custom icons for toolbar actions, better color scheme or dark mode), etc.

Throughout these future developments, the solid foundation we built in Milestone 3.1 will make it easier to integrate new features. The modular structure means, for instance, the network thread can call MainWindow.update\_frame() via a signal without needing to know about UI internals, and the MainWindow can call a method in calibration module when needed, etc.

By following this step-by-step plan, we ensured that at each stage we had a working application (even if partially functional), which helps in catching layout issues or integration problems early. We have also included multiple **test checkpoints** to verify that each component behaves as expected. This reduces the likelihood of large, hard-to-debug issues later on.

*In summary*, Milestone 3.1 achieved the creation of a PyQt GUI application framework for the dual-device controller. We have a main window with all necessary panels and controls laid out and interactive in placeholder form. This provides a clear visual and structural blueprint for implementing the actual experimental control logic in subsequent milestones.

[[1]](https://realpython.com/python-pyqt-layout/#:~:text=,very%20center%20of%20the%20window) PyQt Layouts: Create Professional-Looking GUI Applications – Real Python

<https://realpython.com/python-pyqt-layout/>