Magneto System Overview

C++ Classes

Mainwindow
(mainwindow.h, mainwindow.cpp, mainwindow.ui)

MainWindow is the class responsible for generating the GUI's primary window. The class's members contain objects of the other major software modules (i.e. Physics module, Control module, etc.). Because of this and because most user interaction is processed by MainWindow, it can be considered the central class of the program. Functionalities include

- Providing general system instructions to new users
- Allowing the user to specify user name, save directory, and log filenames
- Connecting the camera and motor controllers to the program
- Allowing for path drawing, loading, saving, clearing, and manipulation/modification
- Setting path-related parameters (interpolation amount, acceptable particle deviation)
- Automating the delivery operation and allowing for manual intervention (pausing, stopping, resuming)
- Controlling widget states out-of and during delivery operation (increases safety)
- Recording system and error messages to the console and to the operation log
- Recording data fields during operation to the data log
- Displaying delivery operation progress updates

MainWindow links the software modules with each other through Qt's signal-slot connections. This allows for proper GUI behavior across classes and for multithreading outside of the primary UI thread (e.g. clicking a button does not interfere with image streaming). Mainwindow.ui contains all the main window widgets and visual components and defines some of their behaviors (e.g. appearance when hovered over, clicked, etc.).

Dialog_SettingsWindow
(dialog_settingswindow.h, dialog_settingswindow.cpp, dialog_settingswindow.ui)

Dialog_SettingsWindow is the class responsible for generating the GUI's settings window. Key functionality of the window includes initiating system calibration (via a simple button click) and allowing for particle detection preview and adjustment (via sliders and corresponding spinboxes to modifies image segmentation parameters). The image segmentation parameters are "Filter Threshold," "Minimum Particle Size," and "Maximum Particle Size." Dialog_SettingsWindow also allows the user to set the hardware command frequency (essentially the number of particle translations per second) and toggle on/off the system axes display resulting from system calibration. Dialog_settingswindow.ui contains all the settings window widgets and visual components and defines some of their behaviors (e.g. appearance when hovered over, clicked, etc.).

OpenCvWorker
(opencvworker.h, opencvworker.cpp)

OpenCvWorker controls all camera image streaming functionality and manages image processing at a higher level. It is one of two classes (alongside ImageSegmentation) that deals heavily with OpenCV library functions. OpenCvWorker is responsible for properly setting up the size of MainWindow's camera viewport display. For image processing, based on signals/functions from classes, OpenCvWorker will feed captured camera frames through certain pipelines (i.e. for system calibration, particle detection, normal streaming, etc.). In these pipelines, the class's ImageSegmentation member performs the low-level image processing. Then, the processed frame is output to MainWindow and GLWidget for proper display.

ControlModule
(controlmodule.h, controlmodule.cpp)

ControlModule effectively controls and automates the particle delivery process from start to finish. One functionality is to perform path discretization using path marker data and path-related parameters from MainWindow. Another is that prior to each particle translation, the class obtains the current particle location from ImageSegmentation, updates its internal data, and determines if the system can proceed with the next translation or the operation needs to be stopped (contains several safety measures). The module also keeps track of checkpoints (path markers) along the delivery path, signaling MainWindow when a checkpoint has been passed. This use of checkpoints allows the user to modify unpassed path markers mid-operation (when the operation is paused) while preventing any modification of already passed markers. This adds much flexibility to user path drawing.

ImageSegmentation
(imagesegmentation.h, imagesegmentation.cpp)

Image Segmentation (.h/.cpp) contains functions for particle and coil detection as well as the coordinate system mapping from pixel space (on an image) to physical space. Particle detection is implemented through a modified background image subtraction process. A "clean" image, where the particle is not in view, is synthesized (see System Calibration Process section below) and is subtracted from all subsequent image frames containing the particle. The particle's location on the image is obtained after a series of morphological operators.

Coil detection is achieved through the implementation of fiducial markers (ArUco) affixed on top of each coil using a 3D printed structure. Each coil marker is unique and can be identified in an image if in view. During the system calibration process (see below section), the coil markers are detected and a cartesian coordinate system is produced from the marker locations; the x-axis by connecting the opposing horizontal coil markers and the y-axis by connecting the opposing vertical coil markers. The intersection of these lines is noted as the origin. Mapping of coordinates from pixel to physical space is accomplished using mathematical transformations (more details can be found here). Field of view (and corresponding distance per pixel) are calculated by comparing the actual measured distance between two coil markers (in mm) to the detected distance (in pixels) in the image stream. Coil markers can be found on the following site: http://chev.me/arucogen/. Specifically, the dictionary used in our application was “4x4 (50,100,250,1000)” and individual marker IDs are: 10, 17, 34, 37 respectively for the +X, -X, +Y, -Y coils.

Physics
(physics.h, physics.cpp)

Physics (.h/.cpp) handles all particle manipulation using neural network or surface fitting model and acts as the main hardware interface for the system. This module implements the weight matrices (after training each neural network) using the Eigen library for matrix arithmetic. Functions to compute and send hardware commands can be found in this module. It’s important to note that Physics only handles hardware communication and computes the required hardware commands to the next point along the path, any decisions relating to path status (on-path, or path completed) are handled elsewhere in the program.

Physics primarily acts as a wrapper class for hardware communication and contains class member objects (roboclaw) that implement methods enabling hardware interfacing. Hardware interfacing is accomplished through a separate set of classes (Roboclaw .h/.cpp, Serial .cc/.h, win .h/.cc). Roboclaw (.h/.cpp) contains the specific commands our motor controllers can understand and Serial,win .h/.cpp enable serial communication through USB. It’s important to mention that the current particle manipulation method is critically dependent on the unique motor controllers used (Roboclaw 2x60A), and the implemented communication method is not guaranteed to work on other motor controllers.

GLWidget
(glwidget.h, glwidget.cpp)

GLWidget is a custom implementation of QOPenGLWidget class. It allows for rendering of OpenGL graphics within a Qt Application. Most of the image rendering workload is allocated to be handled by the GPU rather than the CPU. This cuts down significantly on CPU usage and program memory. Most importantly, the use of OpenGL mitigates visual artifacts (such as Screen Tearing) from streaming at high frame rates. During a delivery operation, GLWidget also draws a “target vector” from the current particle location to the target location along the path. This functions as a visual aid for how well the particle is following the delivery path.

PathPointMarker
(pathpointmarker.h, pathpointmarker.cpp)

PathPointMarker is the graphical representation of a user-drawn path point on the screen (drawn on the QGraphicsScene). Similar to a linked list node, the class contains references to previous and next objects (other PathPointMarkers and their connecting lines). PathPointMarker also contains a PathPtStruct member that holds location data, which is updated whenever the marker is moved on the screen. A PathPointMarker can change its status based on whether the particle has passed it during the delivery operation. This renders the marker immovable, which adds a layer of safety to path modification mid-operation.

GraphicsViewer
(graphicsviewer.h, graphicsviewer.cpp)

GraphicsViewer is a custom implementation of QGraphicsView, a widget used to display the contents of a QGraphicsScene. It is used to display the user-drawn path, including the markers and their connecting lines. GraphicsViewer sends appropriate signals to the MainWindow based on user mouse actions (i.e. left clicking, right clicking, hovering over a PathPointMarker) to enable safe and flexible path drawing/editing.

Point
(point.h)

Point.h includes a "Point" parent class, "PathPoint" child class, and "PathPtStruct" struct. Point and PathPoint are data structures used to represent particle location and the components of the delivery path and are used extensively by most other classes within the program. Both classes handle double precision coordinates and contains a function to compute the euclidean distance between two points. The PathPoint class does not add any new functionality from its parent but is present so that it can be expanded upon for future project iterations. PathPtStruct is used to conveniently contain the physical and pixel coordinate data of a conceptual point. This data structure is key to the ease and flexibility of handling location data throughout the entire program.

Macros
(macros.h)

Macros is used to store convenience macros and functions that would see heavy use throughout the entire program. Currently, only one macro is employed in this iteration of the project, but this can be expanded on for future iterations. The ERROR_FORMAT macro serves to format error messages that will be printed to the console and recorded by the system log. The format provides the class and line number in the code where an error occurred, making it very useful for debugging purposes.

System Calibration Process

After setting up the camera image stream and establishing communication with motor controllers, the user is required to initiate a calibration process to configure particle and coil marker detection as well as prepare the internal physical coordinate system. This process has been optimized to conclude in two distinct phases and requires minimal user interaction (a single button click). The two calibration phases are as follows:

1. **Information Collection**: Send a series of hardware commands to translate the particle in a fixed, cyclic pattern around the petri-dish. Between motor controller commands, capture and store image frames containing the particle and coil markers in view. This process is repeated for a total of 8 hardware commands (two full rotations around the petri-dish).

2. **Post Processing:** To remove the (small) impact of light pollution, each image frame captured in the previous step is segmented for the coil marker locations. The detected location of each coil marker is then averaged across all captured frames and used to initialize the internal coordinate system (see Image Segmentation section above). To synthesize a "clean" image with no particle in view, all stored image frames are averaged. Since the only moving object in frame is the particle, the resulting averaged image will effectively contain no particle in view while all other surroundings are constant (petri-dish, marker structure, etc). This averaged image is stored for particle detection later in the system process (see Image Segmentation section above).

Particle Delivery Process

At the high level, the particle delivery process can be described by 4 overarching steps:

1. **Graphical User Interface (GUI)** continuously receives and streams camera frames. A system clock controls the frequency of particle translation. On every clock timeout, the GUI first checks that all other system components are ready for translation and then signals to the Image Segmentation module to provide the current particle location.

2. **Image Segmentation** continuously tracks the particle by processing streamed camera images. To prepare for translation, the current detected particle location is acquired and converted to physical coordinates. A data check using this location is performed to assess delivery progress along the path. If no system errors are detected, location parameters are passed into the Particle Translation Model.

3. **Particle Translation Model** uses the location parameters to calculate the necessary hardware instructions to perform the desired translation. Using information from previous translations, feedback features are applied to both augment and regulate the model's outputs.

4. **Coil Array Magnetic Control System** executes hardware instructions to run current through connected solenoid coils using motor controllers. Different currents can be applied simultaneously to multiple coils. After successful execution, motor controller status is set as available for the next translation.