P20 – Raspberry Pi Whiteboard

2 Background and Design

2.1 Threads and Mutexes

- Threads allow a single program to run multiple pieces of code at one time, therefore utilizing multiple cores (threads) and making certain tasks easier to implement, e.g. handling multiple clients from one server.
- Pthread is a library which conforms to the IEEE POSIX standard allowing it to work on majority of hardware.
- Qthread is a library in QT which allows implementation of threads.
- Race conditions can occur when multiple threads have access to shared memory. Since the computer swaps between threads (according to a set algorithm) you may not get the results you're expecting since you don't know in which order or when each part of each thread is run.
- Mutexes (mutual exclusion) can be used to avoid thread races. It prevents access of variables from other threads whilst one thread is in the mutex state. This means the entirety of the mutex locked code is run in one go and the computer will not switch threads whilst running in this mutex state.

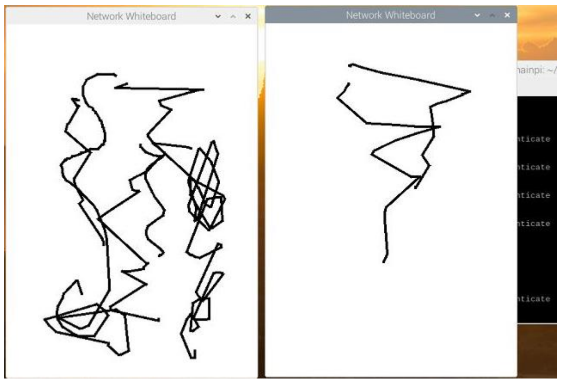
2.2 Application Design

- The send window will have a basic GUI which allows the user to draw in the window. To help save up space and reduce the amount of data sent over the network, I will be using a series of lines which are drawn end to end meaning I only need to send over two coordinates in each network message. This means the resolution of the drawing is determined by the poll rate of the GUI, this may be refined later on to create better looking, more accurate drawings. These lines will be drawn onto a QImage. This will simplify the process of loading and saving images from a file and also allows the currently drawn lines to be kept on the screen. Only part of the image needs to be redrawn which will make the GUI more responsive since less area is updated (faster polling rate).
- As stated previously, the drawings are sent as a series of lines joined end to end. Over the network 2 points will be sent per message and the receiving GUI will then draw that line. This will be sent as the mouse moves, not once the button has been released so the drawing will be sent in real time.

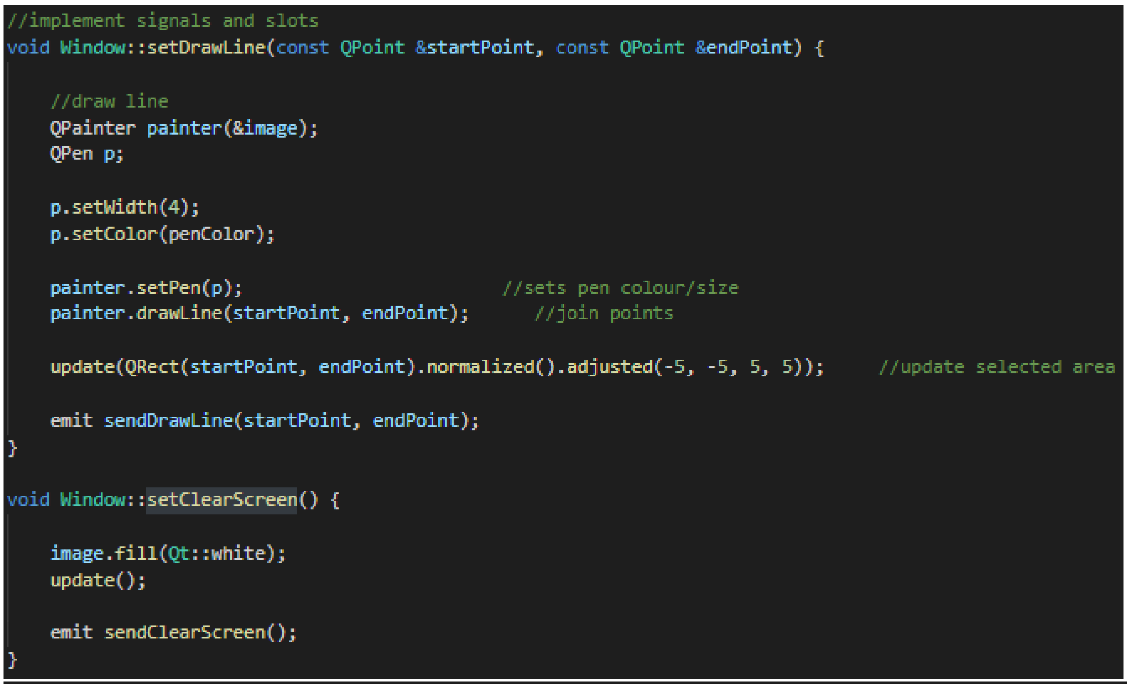
- Since I am using QPoints as the datatype which defines the mouse position, these QPoints use “int” values to store the x and y position. Therefore I will create my own data type which will convert these ints into a series of 4 chars (since one int is 4 bytes in size), therefore 16 char characters will be sent in total (4 for x1, 4 for y1…). In addition to this I will also need to implement the transmission of the pen width and pen colour. The pen width will be another int type converted to 4 chars and pen colour will be 3 lots of char variables. This data will only be sent when the pen colour or width is changed to avoid unnecessary data being sent. This will serialize sufficient data to create an exact duplicate image on the receiving whiteboard.
- Threading will be implemented to aid in both the transmission and reception of information. This allows the GUI to keep updating without it having to wait for the information to come in. This leads to a more responsive UI and is a far more efficient way of doing it. The receiving thread will have a buffer and a flag, the main loop of the program will poll to see if the incoming flag is true. This means the buffer is full and the program can then read the incoming information and draw the relevant information onto the screen.
- For the transmission of the data, I will take inspiration from SPI. So for this I will use a CS bool which will signal the start and the end of the packets, a CLK to clock out each bit and then finally have a data pin DAT which will change on the rising edge of the CLK. The receiving thread will wait for the CS bool to drop and then it will start to read the packets until it is finished, and CS rises again.
- To receive these packets at the receiving end, I will read in the data on the rising edge of the clock, just after the transmitting end unlocks their thread lock. This way I know that all the bits are set to the right value as the transmitting end has definitely finished setting them. At that point it will parse the data and set the relevant flag for the main program to pick up on. With this layout, it allows bi-directional communication if we add a second CS bool (in order to control flow). The main program will take this data and draw the relevant data onto the screen, in the same manner that drawing with the mouse does so that no data is lost and the picture will be stored.

3.1 GUI Send and Receive Windows

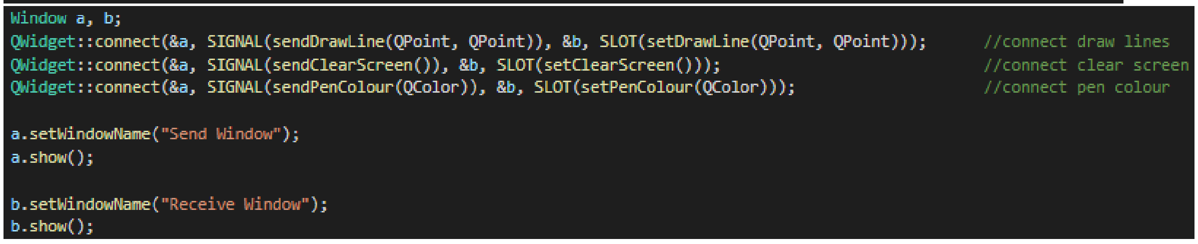
Firstly, I started off by making a simple GUI whiteboard to test that all of the mouse input and QImage parts work. After this I ended up with two separate windows which you can draw on, but no data is passed between them.



After I had verified that this program works, I then moved onto the task of transmitting between windows. The best way to do this is through signals and slots. This simplifies things since all I have to do is create two of the same windows and simply link the functions between them to create one-way communication. I did this by implementing the signal function sendDrawLine and the slot setDrawLine. Each of these functions take two QPoint variables in and then when the setDrawLine is run, it will draw the data on the local window and also trigger the sendDrawLine which when linked properly will send the data over to be drawn on the remote window. Alongside the setDrawLine function, I also implemented functions to set the pen size, pen colour and also to clear the screen. The code for said functions is pictured below.



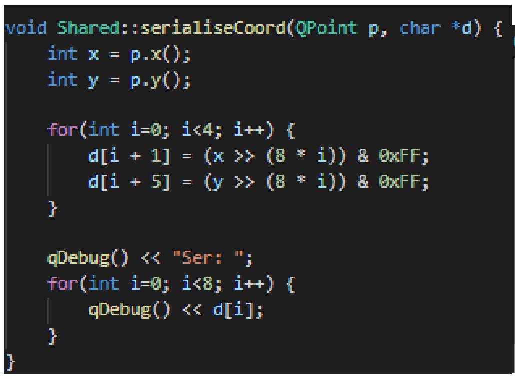
```cpp
//implement signals and slots
void Window::setDrawLine(const QPoint &startPoint, const QPoint &endPoint) {

    //draw line
    QPainter painter(&image);
    QPen p;

    p.setWidth(4);
    p.setColor(penColor);

    painter.setPen(p);                          //sets pen colour/size
    painter.drawLine(startPoint, endPoint);       //join points

    update(QRect(startPoint, endPoint).normalized().adjusted(-5, -5, 5, 5));     //update selected area

    emit sendDrawLine(startPoint, endPoint);
}

void Window::setClearScreen() {

    image.fill(Qt::white);
    update();

    emit sendClearScreen();
}
```



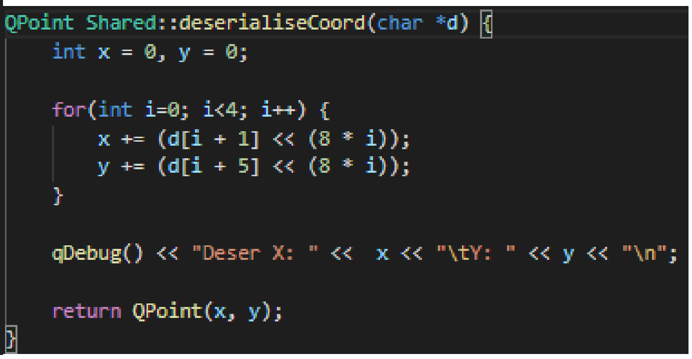
```cpp
Window a, b;
QWidget::connect(&a, SIGNAL(sendDrawLine(QPoint, QPoint)), &b, SLOT(setDrawLine(QPoint, QPoint)));      //connect draw lines
QWidget::connect(&a, SIGNAL(sendClearScreen()), &b, SLOT(setClearScreen()));                            //connect clear screen
QWidget::connect(&a, SIGNAL(sendPenColour(QColor)), &b, SLOT(setPenColour(QColor)));                    //connect pen colour

a.setWindowName("Send Window");
a.show();

b.setWindowName("Receive Window");
b.show();
```

3.2 Serializing and Deserializing Drawing Commands

As discussed in my design, each QPoint coordinate will be converted to 8 x 8-bit values (in this case chars). To do this I created a separate class which will also serve the purpose of acting as the class to hold the shared variables which I will eventually use to connect the two classes. This class named “Shared” will have functions which take a QPoint variable and return an array of 8 chars, another function will then take 8 chars and convert it back into a QPoint. Therefore, I wrote the following functions to serialize and reserialize.

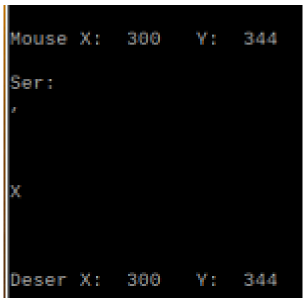


```cpp
void Shared::serialiseCoord(QPoint p, char *d) {
    int x = p.x();
    int y = p.y();

    for(int i=0; i<4; i++) {
        d[i + 1] = (x >> (8 * i)) & 0xFF;
        d[i + 5] = (y >> (8 * i)) & 0xFF;
    }

    qDebug() << "Ser: ";
    for(int i=0; i<8; i++) {
        qDebug() << d[i];
    }
}
```



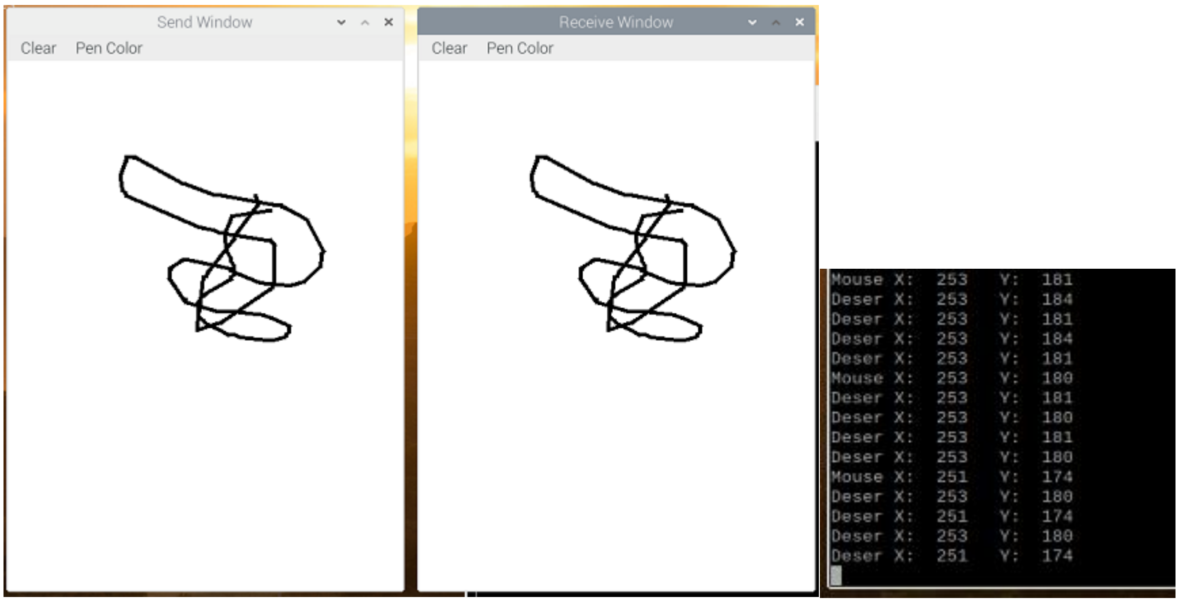
```cpp
QPoint Shared::deserialiseCoord(char *d) {
    int x = 0, y = 0;

    for(int i=0; i<4; i++) {
        x += (d[i + 1] << (8 * i));
        y += (d[i + 5] << (8 * i));
    }

    qDebug() << "Deser X: " <<  x << "\tY: " << y << "\n";

    return QPoint(x, y);
}
```

I then tested this code in the mouseMoveEvent function. I then take the point of where the mouse is, serialize it and then immediately deserialize it to see if the point still matches up.



```
Mouse X:  300   Y:  344

Ser:
,



X


Deser X:  300   Y:  344
```

This was then implemented into my code so that the only method of communication between the two windows was by serialized data. I firstly did this with the drawing points. For this I had a shared set of char arrays which both Window objects could access. Then in the mouse move function I would serialize the data into the shared buffer. Then when the setDrawLine function was run, each Window would deserialize the data in the shared buffer and draw the relevant lines with this data. This can be seen working below, where there is a mirror image on each screen and the debugging console shows the mouse coordinates, followed by each Window deserializing 2 points each.



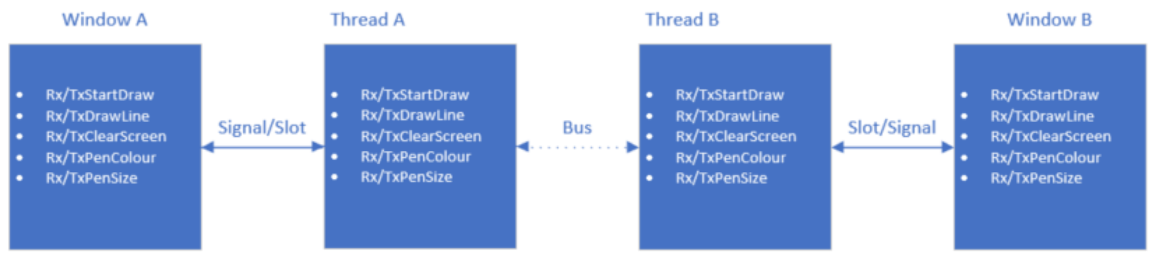
```
Mouse X:  253   Y:  181
Deser X:  253   Y:  184
Deser X:  253   Y:  181
Deser X:  253   Y:  184
Deser X:  253   Y:  181
Mouse X:  253   Y:  180
Deser X:  253   Y:  181
Deser X:  253   Y:  180
Deser X:  253   Y:  181
Deser X:  253   Y:  180
Mouse X:  251   Y:  174
Deser X:  253   Y:  180
Deser X:  251   Y:  174
Deser X:  253   Y:  180
Deser X:  251   Y:  174
```

I then continued by implementing functions to serialize and deserialize other pieces of data like the pen colour and the clear screen commands. As you can see in the functions above, it serializes and deserializes from chars 1 to 9. This makes space for a header byte which will tell us the type of data which is being transmitted. For example, if the header char is ‘x’ then this will be a clear screen command. A ‘p’ will be for a QPoint and ‘c’ will be for a colour change.

3.3 Implementing Send and Receive Threads

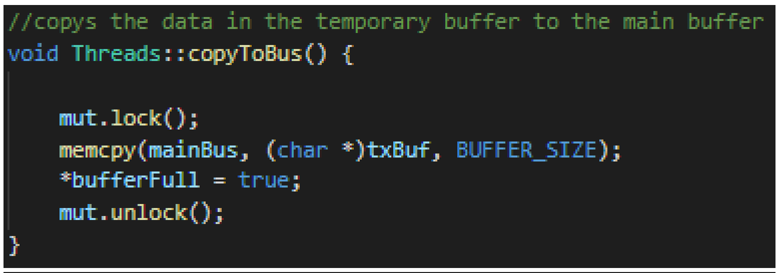
At this point in the project I decided to restructure my code to make threading easier to implement. I decided to re-write all my slots function. I renamed them all rx"…" and the function would draw on the local screen as before but would not emit the signal to be drawn on the other screen. This means in the mouse function or button function I would have to execute the command locally and also emit the signal in the same function. The reason I did this is because it allows me to link the rx slots from one window to the tx signals on the other and vice versa, and because the rx functions do not emit a signal it means you don't get an infinite loop of one calling the other. Also this means I can just implement threads with signals and slots and the threads will simply emit a signal when they receive data and when one of their slots is triggered they will transmit the right data.

With this, I then started the process of implementing threads. To do this all I have to have is each window having its each individual thread which they are connected bidirectionally using signals and slots. Then each thread will handle the transmitting and receiving of data. The topology of how this will work is pictured below.

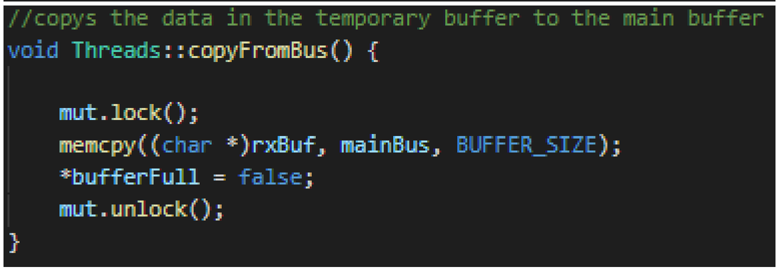


To make implementation of the sending and receiving over the bus, all the incoming data from the window will be stored to a local buffer. This means that the data will be passed over the bus in the thread as soon as possible which keeps the program responsive. This will also help in the implementation of the communication protocol later. Then for the handling of data received over the bus, I have a single function which will be run by the thread when the data is received. This function will deserialize the data and then parse it, emitting the relevant signal depending on the type of data coming in.

With this, I wrote the following functions. Firstly the copyToBus and copyFromBus functions use a QMutex to lock the thread to stop the data from changing whilst reading and writing to the bus. Then it either copies the data too or from the bus to a local buffer and then unlocks the thread.

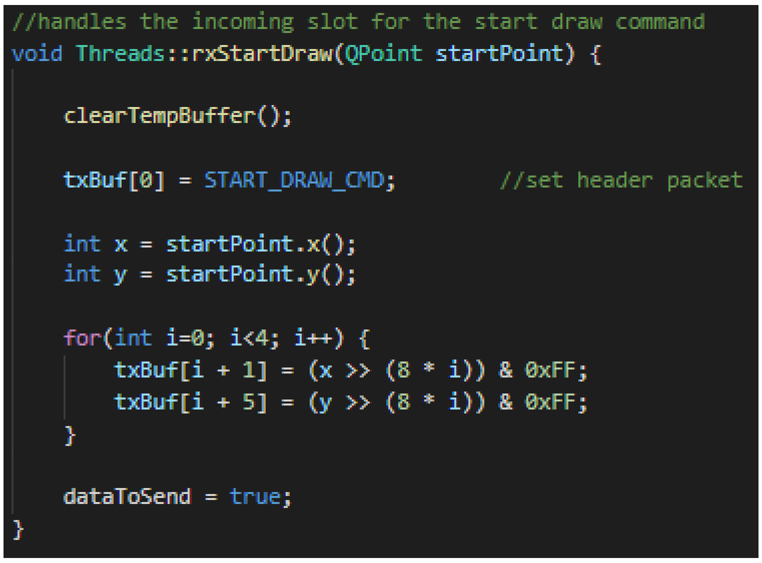


```
//copys the data in the temporary buffer to the main buffer
void Threads::copyToBus() {

    mut.lock();
    memcpy(mainBus, (char *)txBuf, BUFFER_SIZE);
    *bufferFull = true;
    mut.unlock();
}
```



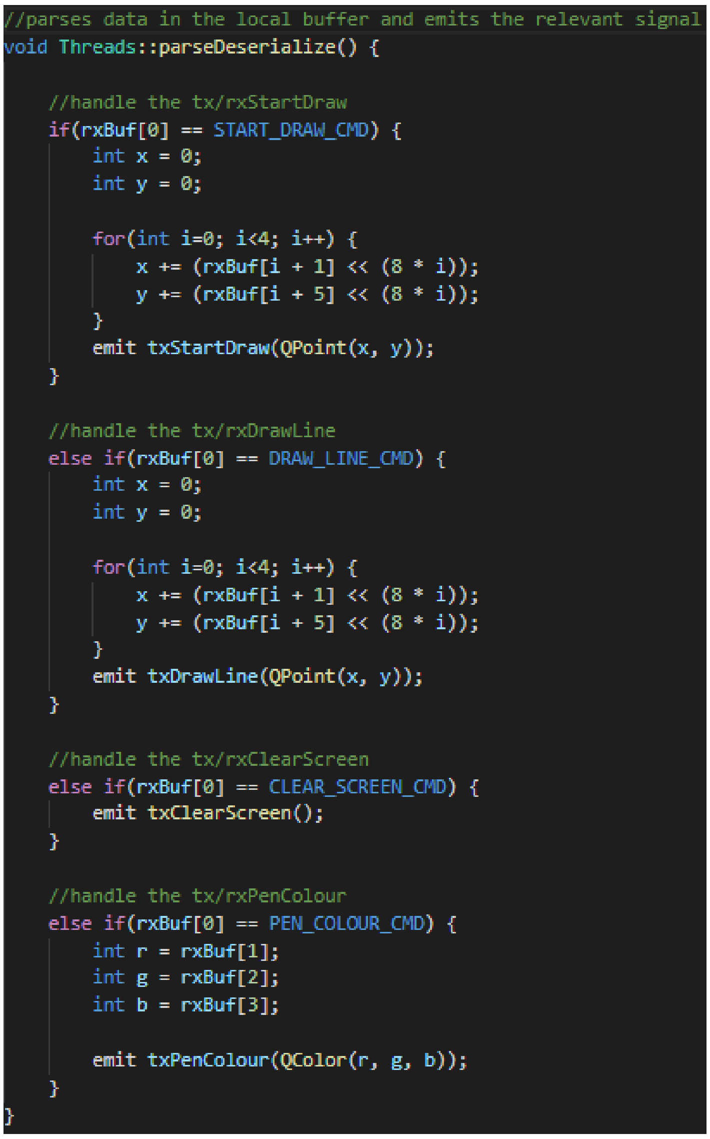
```
//copys the data in the temporary buffer to the main buffer
void Threads::copyFromBus() {

    mut.lock();
    memcpy((char *)rxBuf, mainBus, BUFFER_SIZE);
    *bufferFull = false;
    mut.unlock();
}
```

Then there are also the slots which the main window sends data to. This slot then serializes the data and moves it to a local buffer, which in turn the main thread will move to the main bus when it is free. One example can be seen below, which uses code from the serialiseCoords function which I wrote in part 3.2.



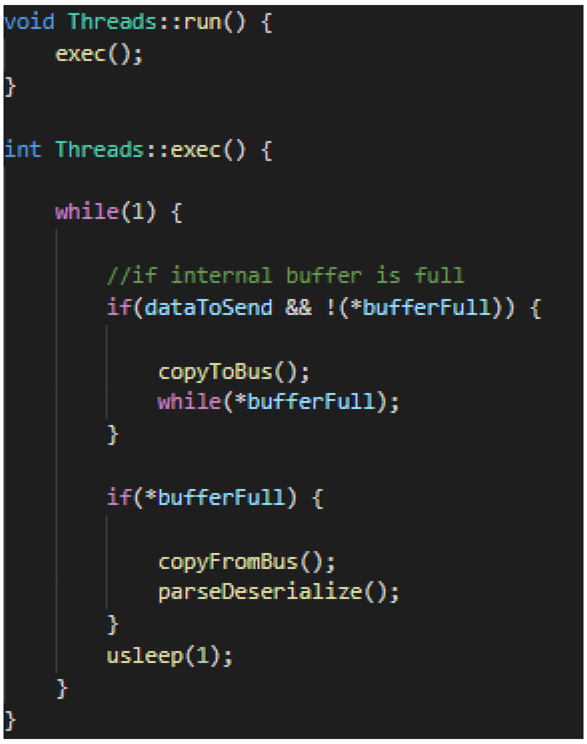
```
//handles the incoming slot for the start draw command
void Threads::rxStartDraw(QPoint startPoint) {

    clearTempBuffer();

    txBuf[0] = START_DRAW_CMD;        //set header packet

    int x = startPoint.x();
    int y = startPoint.y();

    for(int i=0; i<4; i++) {
        txBuf[i + 1] = (x >> (8 * i)) & 0xFF;
        txBuf[i + 5] = (y >> (8 * i)) & 0xFF;
    }

    dataToSend = true;
}
```

I also wrote a function which parses and then deserializes the data in the local buffer. It then emits the relevant signal depending on the data which is presented to it.



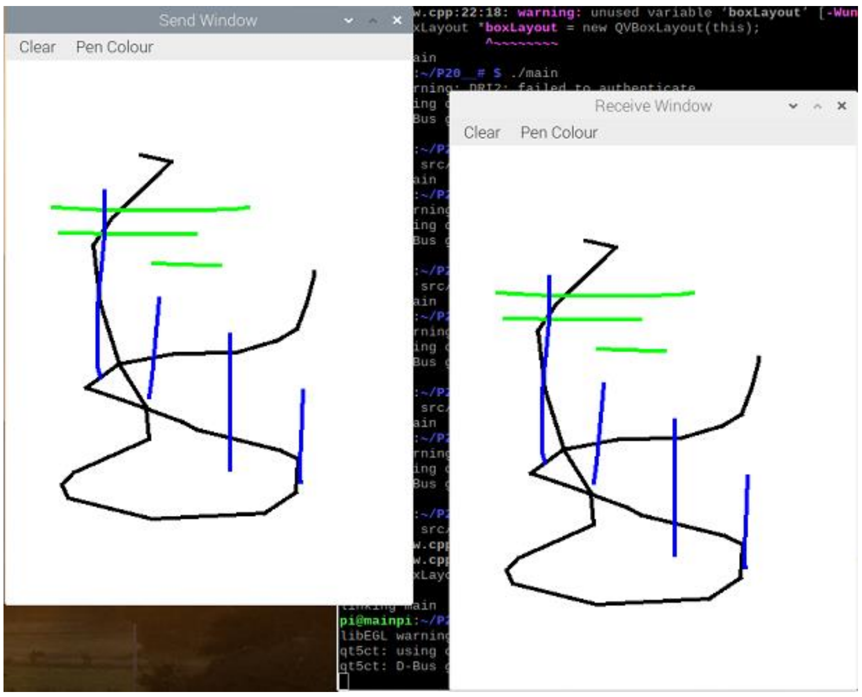
```
//parses data in the local buffer and emits the relevant signal
void Threads::parseDeserialize() {

    //handle the tx/rxStartDraw
    if(rxBuf[0] == START_DRAW_CMD) {
        int x = 0;
        int y = 0;

        for(int i=0; i<4; i++) {
            x += (rxBuf[i + 1] << (8 * i));
            y += (rxBuf[i + 5] << (8 * i));
        }
        emit txStartDraw(QPoint(x, y));
    }

    //handle the tx/rxDrawLine
    else if(rxBuf[0] == DRAW_LINE_CMD) {
        int x = 0;
        int y = 0;

        for(int i=0; i<4; i++) {
            x += (rxBuf[i + 1] << (8 * i));
            y += (rxBuf[i + 5] << (8 * i));
        }
        emit txDrawLine(QPoint(x, y));
    }

    //handle the tx/rxClearScreen
    else if(rxBuf[0] == CLEAR_SCREEN_CMD) {
        emit txClearScreen();
    }

    //handle the tx/rxPenColour
    else if(rxBuf[0] == PEN_COLOUR_CMD) {
        int r = rxBuf[1];
        int g = rxBuf[2];
        int b = rxBuf[3];

        emit txPenColour(QColor(r, g, b));
    }
}
```

Finally there is the main thread loop which handles the moving of data to and from the mainbus. It makes sure there are no collisions and that the data is sent and received properly. This is done using the bufferFull boolean which is a shared pointer between the two threads. When a thread puts data onto the mainbus, it sets the bufferFull to true and when the other thread sees this, it then reads the data and sets bufferFull back to false to acknowledge that it has received the data.



```
void Threads::run() {
    exec();
}

int Threads::exec() {

    while(1) {

        //if internal buffer is full
        if(dataToSend && !(*bufferFull)) {

            copyToBus();
            while(*bufferFull);
        }

        if(*bufferFull) {

            copyFromBus();
            parseDeserialize();
        }
        usleep(1);
    }
}
```

As we can see once the threads and windows are connected via signals and slots, it behaves just as before and is still just as responsive since the communication is handled by threads so does not interfere with the main GUI loop.



```
w.cpp:22:18: warning: unused variable 'boxLayout' [-Wun
xLayout *boxLayout = new QVBoxLayout(this);
                ^~~~~~~~~
ain
:~/P20_# $ ./main
rning: DRI2: failed to authenticate
ing
Bus
:~/P
 src
ain
:~/P
rning
ing
Bus
:~/P
 src
ain
:~/P
rning
ing
Bus
:~/P
 src
ain
:~/P
rning
ing
Bus
:~/P
 src
w.cpp
w.cpp
xLay
linking main
pi@mainpi:~/P
libEGL warning
qt5ct: using
qt5ct: D-Bus
```

3.4 Implementing the Communication Protocol

Due to how I have implemented the threading so far, to implement communication over shared Boolean variables, all I have to do is change the copyToBus and copyFromBus functions. But first I need to decide how the protocol will work.

Firstly, there will be 3 boolean variables. The bufferFull which is already implemented and then mainBus will be replaced by one clock boolean and one data boolean. As before, when one of the threads wants to send data it will pull the bufferFull high, then to acknowledge this the receiving thread will then put the data pin high. This signals the start of the transmission. After this, the transmitting thread will change the data boolean on the rising edge of the clock and the receiving thread will read the data on the falling edge. This makes sure that the data boolean is stable when reading the data. Then to finish of the transmission the bufferFull line is then pulled back to low again and the lines are all set to 0.

I tried implementing this boolean communication but did not manage to get it working. I have included in the final code the working 3.3 code (threads.h, threads.cpp) and then the code for 3.4 (threadsBool.h, threadsBool.cpp).