P20 Lab Report

Mohamed Elrayah

[me3g19@soton.ac.uk](mailto:me3g19@soton.ac.uk)

Mitchell Granger

[mg1g19@soton.ac.uk](mailto:mg1g19@soton.ac.uk)

To detect mouse events, a function was created to detect the left button state, right button state and any other trigger. If the left button is clicked, it is mapped to the ‘*event*’ pointer, which is compared with the ‘*LeftButton*’ from a Qt inherited class. These events are then stored into two-dimensional *vector*s, ‘*x\_temp*’ and ‘*y\_temp*’. If the right button is clicked, the event is compared to the ‘*RightButton’* from a Qt inherited class [1]. The *vector*s ‘*x\_temp*’ and ‘*y\_temp*’ are stored as a new row in the two-dimensional *vector*s ‘*x*’ and ‘*y*’. ‘*x\_temp*’ and ‘*y\_temp*’ are then cleared to allow a new row of coordinates to be created [2]. This has the effect of creating a break in the stream of coordinates and the printing of consecutive lines on the window. This means that individual drawings can be created on the window without connecting lines. If the central scroll-wheel of the mouse is pressed, all four *vector*s are cleared and the screen is updated (within the ‘*clear()*’ function) with the newly empty *vector*s and has the effect of clearing the window.

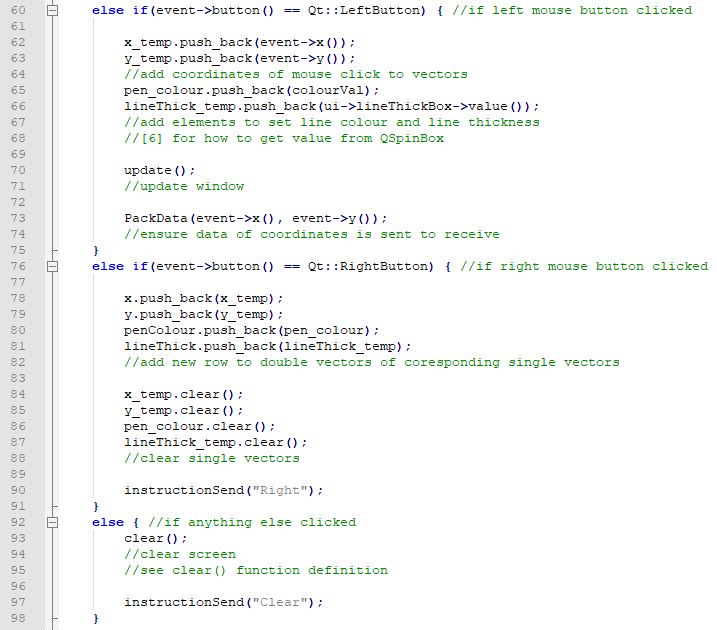


Figure 1

The ‘*mousePressEvent*’ function from the ‘*Transmit*’ class. This shows what happens in response to various mouse events.

In both classes, the coordinates are stored in two pairs of *vector*s for the *x* and *y* coordinates. These are the double *vector*s ‘*x*’ and ‘*y*’ as well as the single *vector*s ‘*x\_temp*’ and ‘*y\_temp*’. New commands are added on to the ‘*x\_temp*’ and *‘y\_temp’* *vector*s, so when the right button is released, these *vector*s are added to their respective double *vector*s before being cleared as shown in Figure 1. This way all the past coordinates are retained with every instruction and lines can be drawn that are separated from each other when desired.

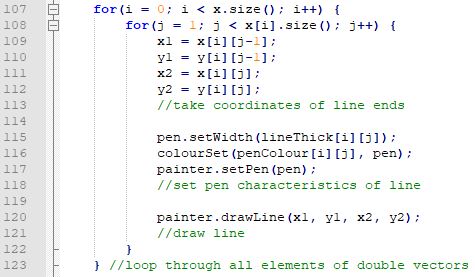


Figure 2

The ‘*paintEvent*’ function runs along with the ‘*mousePressEvent*’ function to draw on the transmit window.

Every time ‘*update()*’ [3] is called, the ‘*paintEvent*’ [4] function prints lines between the consecutive coordinates stored in the *vector*s on the command in the window. The first ‘*for*’ loop in Figure 2 prints according to the two-dimensional *vector*s ‘*x*’ and ‘*y*’ whilst the second ‘*for*’ loop prints according to the *vector*s ‘*x\_temp*’ and ‘*y\_temp*’. In both loops, the ‘*.drawLine(int, int, int, int)*’ function is called using the ‘*QPainter*’ [5] object ‘*painter*’; this paints the lines on the window. An if statement checks if the ‘*x\_temp*’ *vector* is occupied (storing a coordinate). This function is also inherited by the ‘*Receive*’ class in order to display what is being drawn, where the commands are sent intermediate variables containing the *x* and *y* coordinates. Using *vector*s to hold the coordinates and a for loop, means all the coordinates are redrawn each time so the screen “remembers” the previous commands. This means the window retains the diagrams each time it is repainted.

The next section was converting commands into a series of bits stored in an array, at the transmit window, and then converting those bits back into commands at the receive window. In the ‘*Transmit*’ class, two functions were created to convert the new coordinates (‘*int coord*’ on line 270, Figure 3) into a *vector* of binary digits (‘*x\_bin*’). Both ‘*x\_bin*’ and the equivalent ‘*y\_bin*’ are public member variables [6] of the ‘*Transmit*’ class. There is an equivalent function called ‘*ConvToBinY(int coord)*’ which converts the *y* coordinate [7].

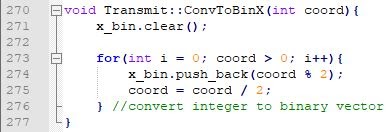


Figure 3

The ‘*ConvToBinX*’ function from the ‘*Transmit*’ class. A protected member function that converts the integer coordinates into a binary *vector* for sending to the ‘*Receive*’ class.

The code inside the ‘*for*’ loop returns a remainder of 1 or 0 (binary digits) which are saved as the latest element of the ‘*x\_bin*’ *vector* before being divided again by 2 with the result being saved back into itself. The ‘*for*’ loop continues the process until the ‘*coord*’ variable becomes 0. This process ensures that the binary packet is returned as a *vector* consisting of a stream of 1s and 0s.

The function in Figure 4 is the opposite of that in Figure 3. This function receives the *vector* created in Figure 3 and converts it back into an integer so that it can be used in the ‘*Receive*’ class for the receive window. This is effectively deserializing the received data packets [7]. The result for each value of the binary digit is added to the ‘*decimal*’ variable so that we eventually get the original integer that we had in the ‘*Transmit*’ class. The ‘*decimal*’ variable is then added as the last element of the ‘*x\_temp*’ *vector* in the *‘Receive’* class. In the ‘*unPackData*’ function, that calls the ‘*ConvToDecX*’ function and the equivalent ‘*ConvToDecY*’ function, ‘*update()*’ is called so that the receive window shows the same as the transmit window.

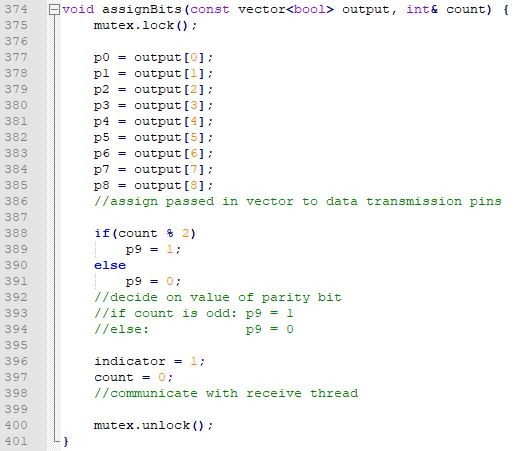


Figure 5

The ‘*assignBits*’ function used in the transmit thread to save a *bool* *vector* on the series of nine bits before setting the parity bit.

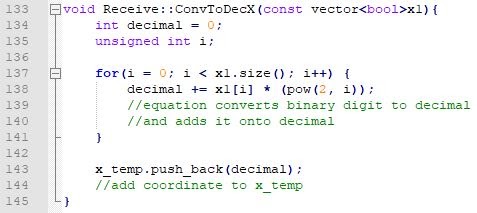


Figure 4

The ‘*ConvToDecX*’ function from the ‘*Receive*’ class. A protected member function that converts a binary coordinate in the form of a *vector* to an integer to be added to the *vector* of *x* coordinates.

The ‘*assignBits*’ function in Figure 5 is what is used by the transmit thread to send the new coordinate or instruction by assigning the bits to nine intermediate variables, representing the wires. By sending the *vector* packet in one go by assigning it to nine variables we were able to make it clearer within each thread the data that was being sent. Prior to creating this code, we had attempted to send the packets one bits at a time but the major flaw with this was that it became confusing and difficult to debug. Furthermore, data transmission would likely have taken too long.

The logic behind using nine bits to send the data with, even though the binary value may not be nine bits long, was also one of practicality as it meant that the receive thread would already know the length of the packet.

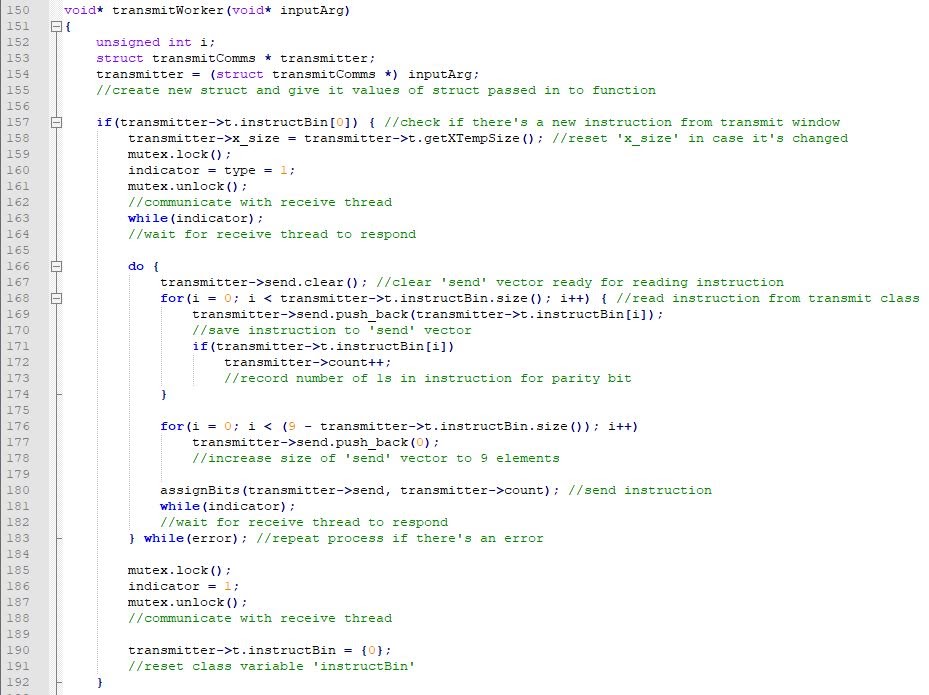


Figure 6

The section of the transmit thread that handles a new instruction and sends the binary value to the receive thread.

It is ensured that no instructions or new coordinates are missed from the transmit window by using one intermediate variable or wire – ‘*wait*’ – to prevent the transmit thread [8] from accepting any new inputs before the receive thread has completed its tasks. This is achieved by using a ‘*while*’ loop that holds the transmit thread in a position of inactivity for as long as ‘*wait*’ is *True* – ‘*wait*’ is converted back to *False* at the end of the receive thread. In order to prevent this pause in the thread from allowing multiple coordinates or instructions to be passed into the ‘*Transmit*’ class before the threads can accept them, therefore resulting in some commands being missed, a ‘*wait*’ variable was added to the ‘*Transmit*’ class’ list of private variables. This is set once the transmit thread detects new data and is reset once the receive thread indicates that it is completed using the ‘*setWait*’ public function of the ‘*Transmit*’ class to stop the ‘*mousePressEvent*’ [1] function and prevent a new command from being accepted. By setting the class ‘*wait*’ variable after a new command is detected and only setting it again after the receive thread has reset the global ‘*wait*’, we ensure that the only time that a new command can only be accepted from the transmit window is when both of the threads are at their final ‘*while*’ loops, waiting for new data. Otherwise, it prints a message through *qDebug* [8]to alert the user of the fact. Because of this wait feature, there is no need for buffering packets at either end of the transmission.

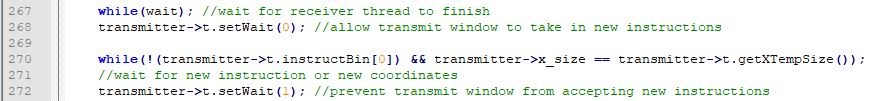


Figure 7

The use of the ‘*setWait*’ function in the transmit thread to communicate back to the *‘Transmit’* class. Also, the ‘*while*’ loop used on line 267 to make the transmit thread wait for the receivethread.

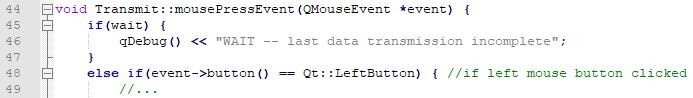


Figure 8

The use of the private ‘*wait*’ variable of the *‘Transmit’* class to prevent the transmit window from accepting a new command if the transmission of the last instruction has not been completed.

Both thread functions take in a *struct* containing their specific variables (which contain the transmitted data packets). As can be seen in Figure 9, this *struct* is saved to a newly created *struct* so that the existing state of the variables is maintained [9].

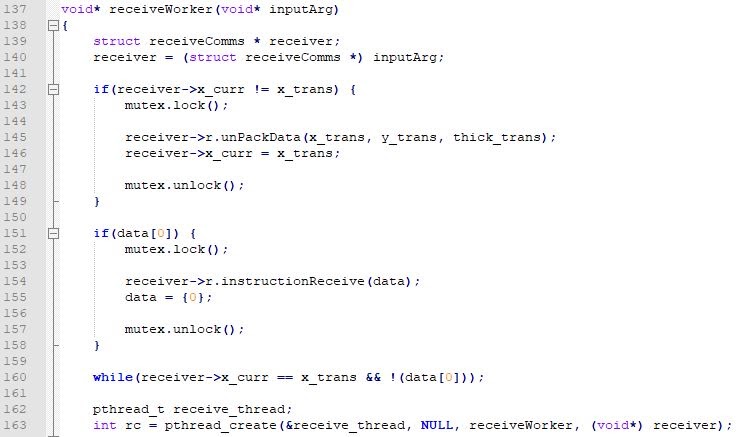


Figure 9

The section of the receive *pthread* that checks for and processes the input from the intermediate global variables of the main program.

Originally – when our code used a combination of intermediate vectors, the first ‘*if*’ statement in the receive thread (on line 142, Figure 9) tests whether the value of the intermediate *vector* ‘*x\_trans*’ has changed by testing it against the *struct vector* ‘*x\_curr*’. The second if function tests the first element of the intermediate *vector* ‘*data*’. This is checking for the same occurrence as with the second ‘*if*’ statement in Figure 6 and the variable follows the same principle - a 1 designates a change and a 0 designates that no action should be taken. The ‘*while*’ loop also performs the same functionality as with the code in Figure 6, however, this time it tests the intermediate variables ‘*x\_trans*’ and ‘*data*’ rather than member variables of the ‘*Transmit*’ class. Once one of the conditions in this while loop is not met, the function can pass it and complete the subsequent commands – it creates a new thread passing in the same function before exiting and ending the thread.

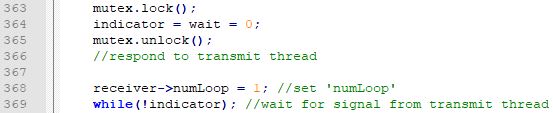


Figure 10

The new conclusion to the receive thread where the ‘*wait*’ bit is set and the ‘*indicator*’ bit is now tested in the ‘*while*’ loop.

With the current system of intermediate *bool* variables being used to transmit the data one value at a time, the pair of ‘*if*’ statements in Figure 9 now tests the *bool* ‘*type*’ to differentiate between an instruction and a coordinate. However, despite the changes in transmission and the fact that only one value can be read at a time, little has changed with the principle operation of the receive thread function. Another key difference between the code in Figure 10 and it’s equivalent in Figure 9 is that the ‘*indicator*’ bit is now tested in the *‘while*’ loop since the same tests can no longer be performed by the receive thread with the new intermediate variables. It can also be noted how the ‘*wait*’ variable is reset before the code reaches the ‘*while*’ loop to indicate that new commands can now be accepted.

As previously mentioned, *bool* variables were eventually used to represent the pins on a Raspberry Pi. These intermediate variables are essential as they represent the wires between the receive and *‘Transmit’* class.

The pins assigned were not only for data transmission purposes, but also included pins for the purposes of checking for transmission errors (‘*error*’), communicate the type of data transmitted (‘*type*’), notify when command completed (‘*indicator*’), ensuring a command is executed before a new one is processed (‘*wait*’) and a parity bit (‘*p9*’). The parity bit works by adding a 1 or a 0 to each stream of 9 binary digits (representing the command) to ensure even parity is maintained [10]. The ‘*indicator*’ variable is altered by both threads when an action is completed and triggers the opposing thread to continue with its code, i.e. signals when the bits are ready to read. The transmit thread sets it to a 1whilst the receive thread resets it to 0. This ensures reliable communication.

Regardless of whether it’s a coordinate or instruction that’s being transmitted, the receive thread calls the ‘*readBits*’ function in Figure 11to save the data to the appropriate *vector*. After all of the data has been read by the Receive thread, the *vector(s)* are passed into a class function which processes the data and performs the appropriate action. For the coordinates, this involves being converted to a decimal using the function in Figure 4. For the instructions and line thickness there are functions that are called to process the data and perform the appropriate action using *if-else* statements. Where appropriate there is an ‘*update()*’ [5] command in these functions that ensure the new information is represented on the receive window and that it displays the same as the transmit window.

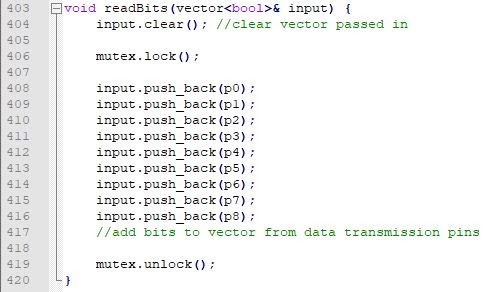


Figure 11

The ‘*readBits*’ function takes in a *vector* from the receive thread and assigns the newly transmitted *vector* to it.

The data in the *‘Receive’* class is retained and stored in the same way as the *‘Transmit’* class: by using pairs of single and double *vector*s. The same goes for many of the functions in the *‘Receive’* class, such as the ‘*paintEvent*’ [4] function, which has only very minor differences to its counterpart in the *‘Transmit’* class, for example the lack of the ‘*wait*’ variable. This means that the methods for drawing on the window are the same as in the *‘Transmit’* class, i.e. using a *vector* to “remember” the coordinates so they are used to re-paint the window each time ensuring no diagrams are lost.

It can be seen in Figure 5, Figure 6, Figure 9, Figure 10 and Figure 11 that the threads [8] use a *QMutex* [11]identified as ‘*mutex*’. This is used in the ‘*if*’ statements, where the values of the intermediate variables are either read from or changed. The *QMutex* is used to ensure that only one of the threads can access the variable at a time. This ensures that one thread isn’t reading or changing a variable whilst the other is simultaneously reading or changing the same variable. If ‘*mutex*’ didn’t lock the variables for each thread, it could result in a mixture of different code being performed simultaneously from both threads which could result in strange data transmissions and errors between the transmit and receive windows. The ‘*mutex*’ isn’t used to lock and unlock the variables during the ‘*while*’ loops because this would lead to them being almost permanently locked and could result in problems or long delays with transferring the data. Furthermore, the two threads accessing the variables at the same time in the context of these tests won’t cause an issue because it can only happen if they are both only reading the variable as a condition for a ‘*while*’ loop or an ‘*if*’ statement since ‘*mutex*’ is used to lock the variables in every other circumstance.

# References

|  |  |
| --- | --- |
| [1] | "QMouseEvent Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qmouseevent.html. [Accessed 2020]. |
| [2] | "std::vector," cplusplus.com, [Online]. Available: http://www.cplusplus.com/reference/vector/vector/. [Accessed 2020]. |
| [3] | "QWidget Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qwidget.html. [Accessed 2020]. |
| [4] | "QPaintEvent Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qpaintevent.html. [Accessed 2020]. |
| [5] | "QPainter Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qpainter.html. [Accessed 2020]. |
| [6] | "Access Modifiers in C++," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/access-modifiers-in-c/. [Accessed 2020]. |
| [7] | "C++ Program to Convert Binary Number to Decimal and vice-versa," programiz, [Online]. Available: https://www.programiz.com/cpp-programming/examples/binary-decimal-convert . [Accessed 2020]. |
| [8] | "C++ Multithreading," tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp\_multithreading.htm. [Accessed 2020]. |
| [9] | "QDebug Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qdebug.html. [Accessed 2020]. |
| [10] | "Data Structures," cplusplus.com, [Online]. Available: http://www.cplusplus.com/doc/tutorial/structures/. [Accessed 2020]. |
| [11] | "Parity Bit," Tech Terms, [Online]. Available: https://techterms.com/definition/parity\_bit. [Accessed 2020]. |
| [12] | "QMutex Class," Qt, [Online]. Available: https://doc.qt.io/qt-5/qmutex.html. [Accessed 2020]. |