# Lab P20: Advanced Programming Design Exercise

## Whiteboard chat for the Raspberry Pi

### Lab Preparation

#### 2.1: Threads and Mutexes

A thread is an independent execution sequence within a single process. They can be used to carry out individual functions in a program in parallel. They exist in the same process and therefore share the same data.

Using the thread library, an object can be spawned to carry out the specified function with a given argument as with the example below:

thread myThread (myFunction, arg1, arg2, arg3);

A mutex; short for mutual exclusion; is an object in C used to prevent the access and modification of shared resources simultaneously. When multiple threads access one piece of data and at least one of them does a write operation, an event known as a race operation occurs which often leads to data corruption or system failure. Implementing a mutex helps solve this using the lock and unlock functions present in it, an example below:

std::mutex random\_mutex;

random\_mutex.lock();

{carry out operation here}

random\_mutex.unlock():

#### 2.2: Using the GPIO Interface

As done in P7, using the GPIO interface is done by calling the wiringPi.h library in the Raspberry Pi,

#include <wiringPi.h>

wiringPiSetup();

pinMode(0, OUTPUT);

digitalWrite(0, HIGH); delay(500) ;

digitalWrite(0, LOW); delay(500) ;

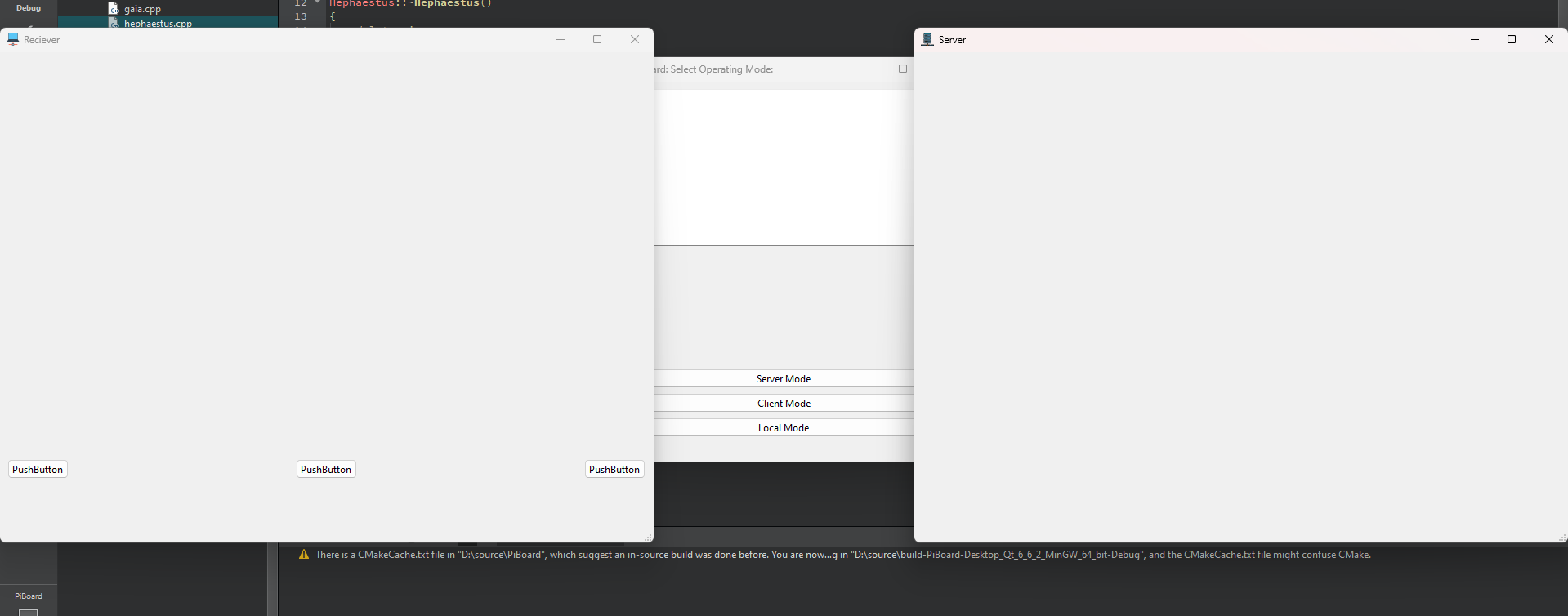
#### 2.3: Designing the application

To make a display window whereby the lines being drawn are retained when being repainted, it is likely that we will have to use a data structure to hold the values for what has been drawn and the position of the drawing relative to the window.

### Actual Lab Work

#### 3.1 The GUI send and receive windows

Two windows were made; one for send and another for receive.



To capture the drawing aspect, the QMuoseEvent() was used. To capture different points of a line, mouseReleaseEvent() was used to identify the first point and the second point, which is later used in painting the window.

For a continuous line which requires constant reading of the mouse position, [mouseMoveEvent](https://doc.qt.io/qt-6/qwidget.html#mousePressEvent)() was used.

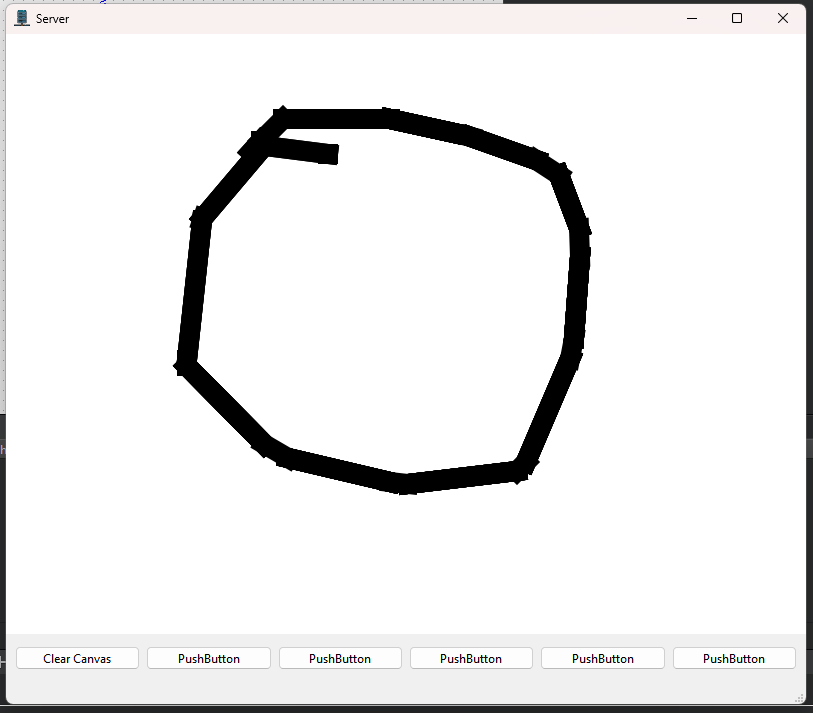


To draw the lines, an image was created using QImage using:

image = new QImage(600, 600, QImage::Format\_RGB32);

image->fill(Qt::white);

An issue was encountered when trying to draw individual points on the canvas. The lines would only be straight and not allow for custom lines, shapes or anything else as shown below.



It seemed that it wasn’t recording the movement while moving the mouse. To fix this, a new class for drawing points was introduced as follows:

void Hermes::drawOnCanvas(QPainter& painter, QPen& pen, int drawMode) {

switch (drawMode) {

case 0:

draw->drawLine(painter, pen);

break;

case 1:

draw->drawPoint(painter, pen);

break;

default:

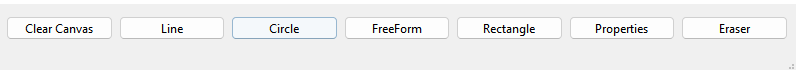
qDebug() << "Unexpected drawMode value:" << drawMode;

qWarning() << "Unexpected drawMode value:" << drawMode;

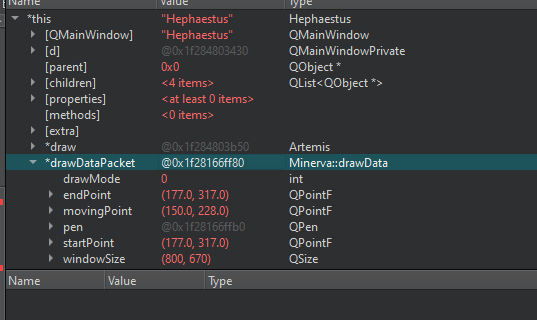
break;

}

Different buttons for different functions such as drawing circles, rectangles, points and erasing were introduced for easier usability.



We have tried initializing the data class in a different class object so as to pass it into the receive and send clients as arguments to have a common shared resource but experiencing an issue where the receiver client is not getting updated (likely) because the shared pointer is accessible, and the data is visible in debug mode:

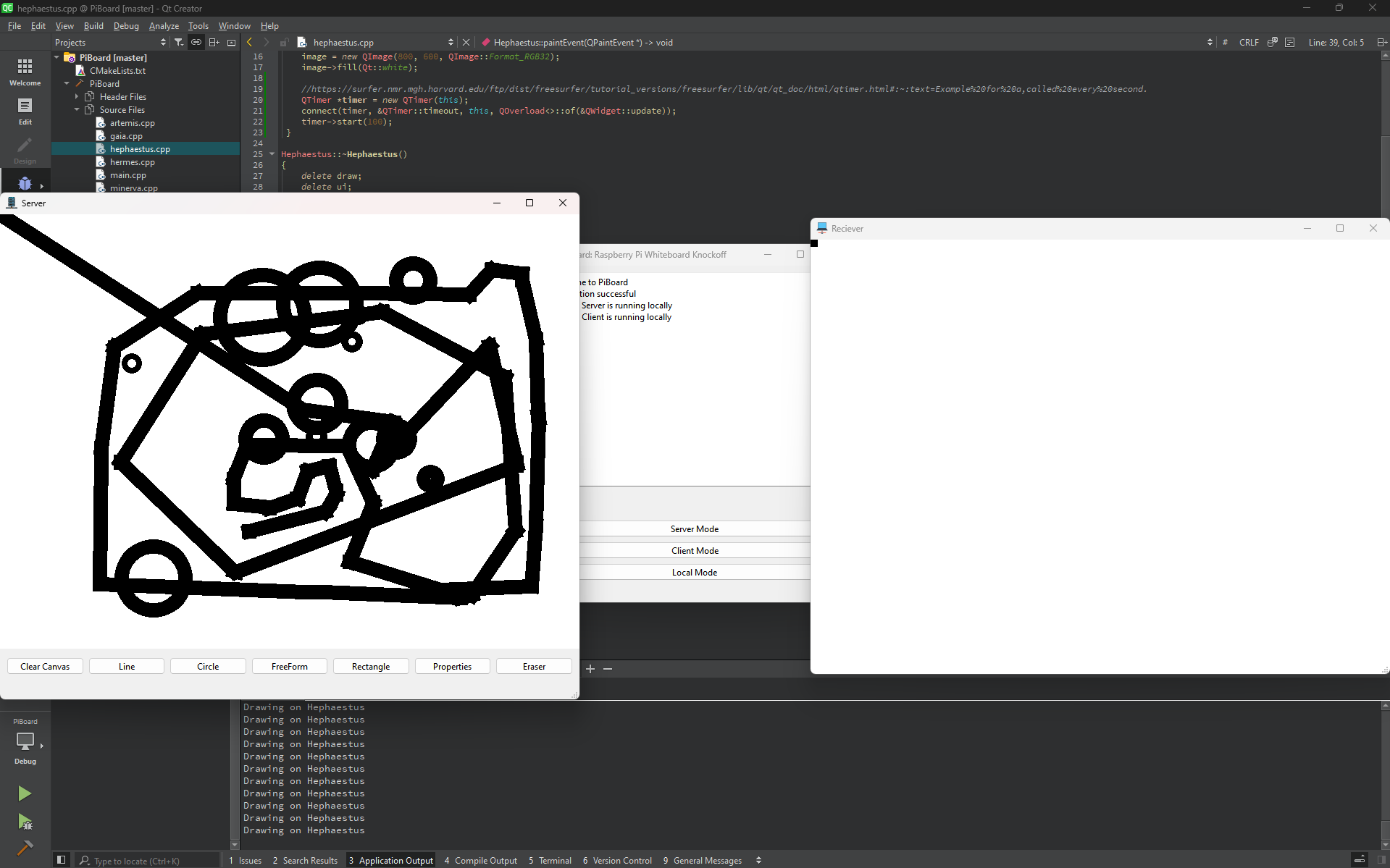


This is the data in debug mode from Hephaestus (Receiving client) having the appropriate start point, end point and so on.

We will try constantly updating the receive window manually using QTimer every 100ms to check for any difference.

<https://surfer.nmr.mgh.harvard.edu/ftp/dist/freesurfer/tutorial_versions/freesurfer/lib/qt/qt_doc/html/qtimer.html#:~:text=Example%20for%20a,called%20every%20second>.

This has not helped as seen below:



This was apparently an issue due to the position variables in the client not being passed through to Artemis ( the drawer class ) and was fixed with the following :

void Hephaestus::drawOnCanvas(QPainter& painter, QPen& pen, int drawMode) {

draw->startPoint = drawDataPacket->startPoint;

draw->endPoint = drawDataPacket->endPoint;

draw->movingPoints = drawDataPacket->movingPoint;

switch (drawMode) {

case 0:

draw->drawLine(painter, pen);

break;

case 1:

draw->drawPoint(painter, pen);

break;

case 2:

draw->drawCircle(painter, pen);

break;

case 3:

draw->drawRectangle(painter, pen);

break;

case 8:

pen.setColor(Qt::white);

draw->erasePoint(painter, pen);

break;

default:

qDebug() << "Unexpected drawMode value:" << drawMode;

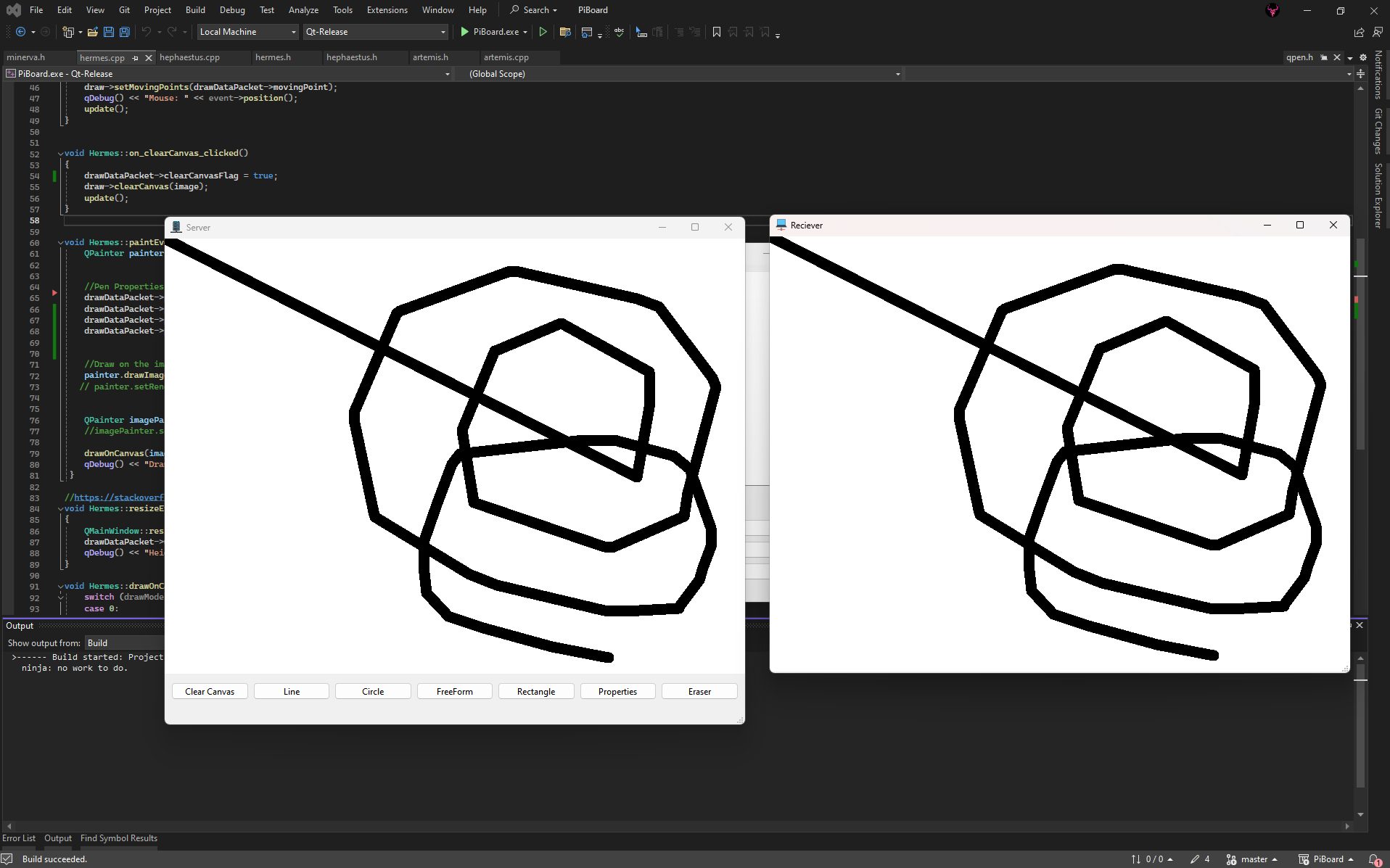
qWarning() << "Unexpected drawMode value:" << drawMode;

break;

}

}

Proof of work in the next page:



To clear the canvas, a global flag was set that is constantly checked on the receiver, and set to false once cleared.

#### 3.2 Serialization of the commands

To serialize the data, we have settled on using the boost library for object serialization following this tutorial.

The objective is to serialize the data packet made from all the attributes used for drawing and instead of passing it as a variable, it will be reconstructed by Hephaestus as sent by Hermes.

<https://www.codeguru.com/cplusplus/an-introduction-to-object-serialization-in-c/>

<https://isocpp.org/wiki/faq/serialization#serialize-simple-types>

Serialization is a way to convert an object in c, in this case, the data packet into a sequence of bytes which hopefully can be transferred over the wires.

The data could then be deserialized in Hephaestus, reconstructing the data needed for drawing.

To serialize the data, instead of using Boost, we opted for QDataStream as boost wasn’t installing at all on windows (what we used for debugging (for faster compile)) and it proves a success.

The following code was used:

//https://doc.qt.io/qt-6/qdatastream.html

void Minerva::encodeData(){

QFile file("file.dat");

if (!file.exists()) {

qDebug() << "File does not exist";

}

file.open(QIODevice::WriteOnly);

QDataStream out(&file);

out << drawDataPacket->startPoint;

out << drawDataPacket->endPoint;

out << drawDataPacket->movingPoint;

out << drawDataPacket->pen;

out << drawDataPacket->drawMode;

out << drawDataPacket->windowSize;

out << drawDataPacket->clearCanvasFlag;

}

void Minerva::decodeData(){

QFile file("file.dat");

if (!file.exists()) {

qDebug() << "File does not exist";

}

file.open(QIODevice::ReadOnly);

QDataStream in(&file);

in >> drawDataPacket2->startPoint;

in >> drawDataPacket2->endPoint;

in >> drawDataPacket2->movingPoint;

in >> drawDataPacket2->pen;

in >> drawDataPacket2->drawMode;

in >> drawDataPacket2->windowSize;

in >> drawDataPacket2->clearCanvasFlag;

}

#### 3.3: Implement Send and Receive threads

We will be using QThreads from these docs:

<https://doc.qt.io/qt-6/qthread.html>

<https://www.geeksforgeeks.org/implement-thread-safe-queue-in-c/>

We have implemented a custom QThreads, inheriting from it and used “Workers” to call the function for sending and receiving the data from the two separate threads.

#pragma once

#include <QObject>

#include <QDebug>

#include "EpimetheusThread.h"

#include "minerva.h"

class Epimetheus : public QObject

{

Q\_OBJECT

public:

Epimetheus(Minerva\* minervaIn);

~Epimetheus();

void receiveDataUsingThread();

void stop();

private:

Minerva\* minerva;

bool t\_stop = false;

};

#include "Epimetheus.h"

Epimetheus::Epimetheus(Minerva\* minervaIn) : minerva(minervaIn) {

}

Epimetheus::~Epimetheus()

{}

void Epimetheus::receiveDataUsingThread() {

minerva->runReceiveThread();

}

void Epimetheus::stop() {

t\_stop = true;

}

We did try using <thread> but had some issues with the timing and synchronization and ended up using QThreads which is platform independent and used <pthread> which is what is recommended by the lab preparation.

We have used mutexes to lock the data structure from being modified when the thread is running as follows: (highlighted in red)

//Serialize Big Packet

QDataStream bigStreamSender(&bigData, QDataStream::WriteOnly);

//Lock data when sending

sendLock->lock();

bigStreamSender << sendDataPacket->startPoint;

bigStreamSender << sendDataPacket->movingPoint;

bigStreamSender << sendDataPacket->endPoint;

bigStreamSender << sendDataPacket->clearCanvasFlag;

bigStreamSender << sendDataPacket->drawMode;

bigStreamSender << sendDataPacket->pen;

bigStreamSender << sendDataPacket->windowSize;

sendLock->unlock();

//unlock data when sending

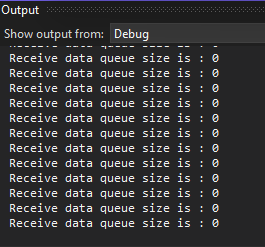
//Queueing big packet

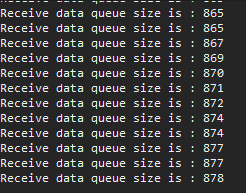
dataQueue.enqueue(bigData);

qDebug() << "Send data queue size is :" << dataQueue.size();

This allows for thread safety, as only one thread is allowed to write to the DataStream (though not needed as we are updating this sequentially)

For the moment, we’re finding out that the queue on the receiver thread is piling up more than usual and I think that’s because it’s sending more than it can receive – a limitation we’re not too sure how to handle.

Without sending anything: Sending and receiving:

After closing of the sender, it is cleared appropriately, we will figure this out later.

#### 3.4: Implementing the communication protocol

We have made a test DMA (direct memory access) – not exactly.

This uses shared variables on both the send and receive those checks if the sent Boolean matches the received Boolean, a function quite simple. (ingles)

void Gaia::testDMA(){

qDebug() << "Button has been clicked";

int pinNumber = 8;

bool sent = true;

bool rec = false;

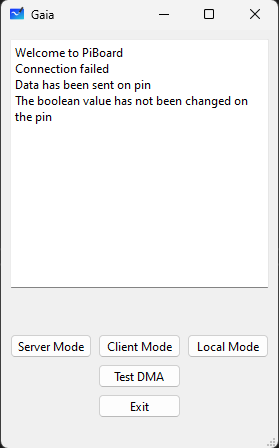
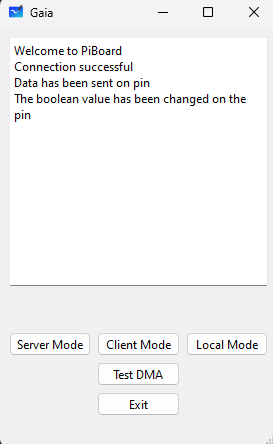
minervaOut->sendBit(8,sent);

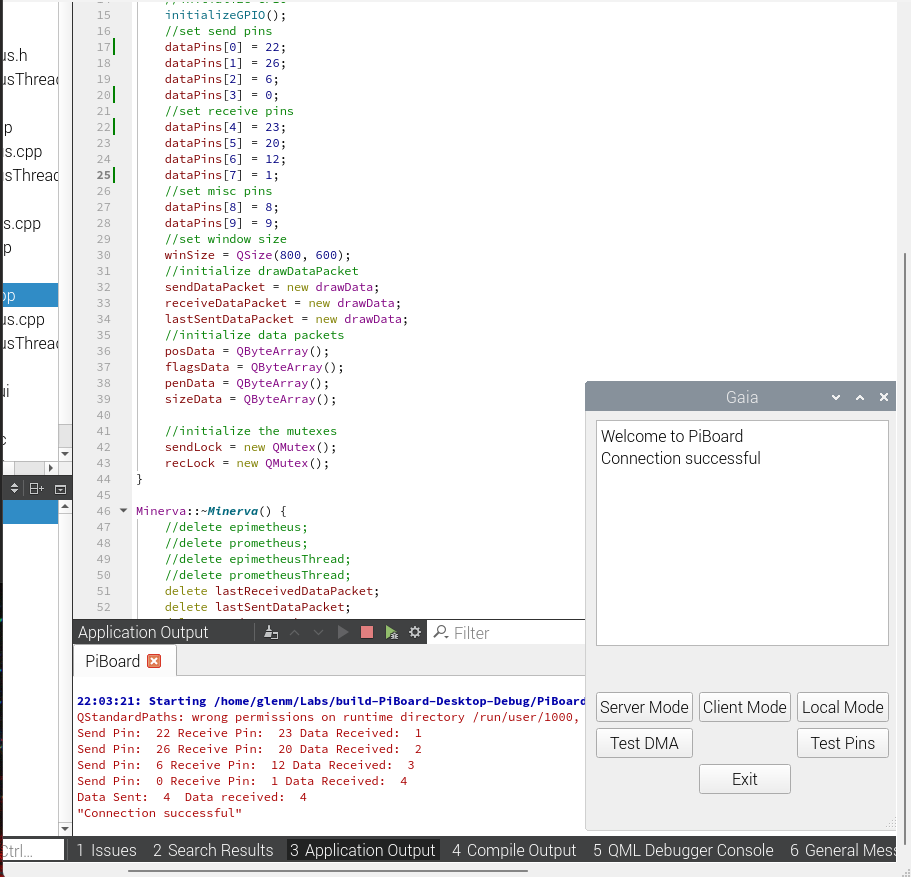
ui->minervaStatus->append("Data has been sent on pin");

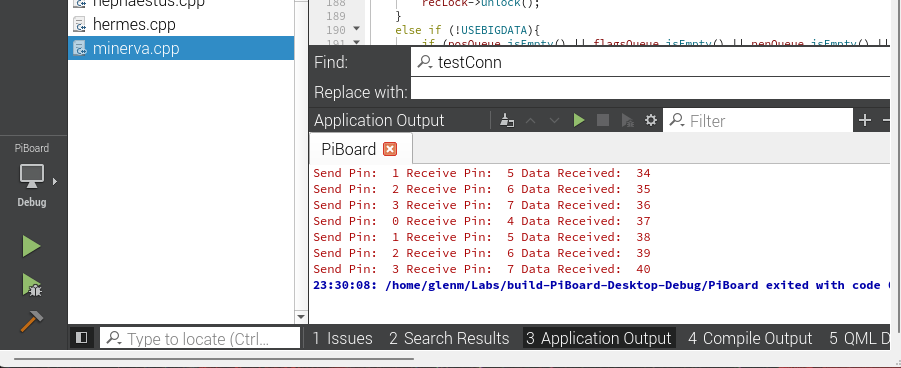
rec = minervaIn->receiveBit(8);

ui->minervaStatus->append(rec ?"The boolean value has been changed on the pin" :"The boolean value has not been changed on the pin");

}

This gives the following result using fakeWiringPi.h – on windows – to simulate actual connection.

The testing of connection was successful as shown below: 



#### 3.5: Read and Write to Physical GPIO Pins

For reading and writing to the physical GPIO pins, the above function was expanded on and it included a send ready function for both the client and receiver. This allows them to only send data when ready to send or receive.

The two Pis are connected to each other using 4 pins for data transfer and 3 pins for synchronization.

We are currently only using 1 pin for testing and will use more when a working version is achieved.

For now, the send ready and receive ready signals do work. There are timeouts implemented if they are not working.

The code used is as follows:

void Minerva::sendReady(bool value , int pin) {

digitalWrite(syncPins[pin], value);

}

bool Minerva::isReceiveReady(int pin) {

return digitalRead(syncPins[pin]);

}

void Minerva::receive() {

int numRetries = 0;

bool senderReady = false;

sendReady(true, 7); // Send the ready signal to pin 7 (GPIO 13) - pin 3 (GPIO 17) receives it

while (!senderReady && numRetries < MAX\_SYNC\_RETRIES) {

delayMicroseconds(SYNC\_TIMEOUT);

senderReady = isReceiveReady(4); // Read the ready signal from pin 4 (GPIO 14) - pin 0 (GPIO 2) sends it

numRetries++;

}

if (senderReady) {

// Sender is ready, acknowledge the ready signal

sendAcknowledgement(true);

// Receive the data

if (USEBIGDATA) {

receiveBigData();

}

else {

receiveMultipleData();

}

sendReady(false, 7); // Send the not ready signal to pin 7 (GPIO 13) - pin 3 (GPIO 17) receives it

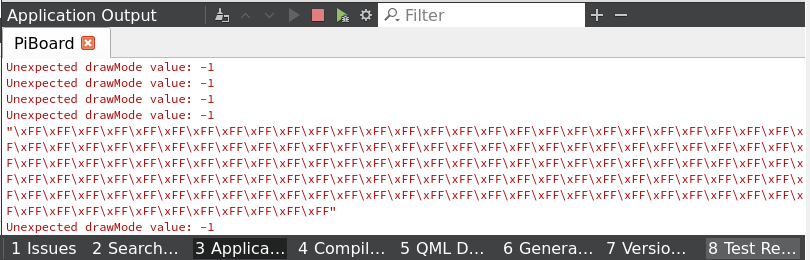
}

else {

qWarning() << "Minerva::receive(): Sender not ready after" << MAX\_SYNC\_RETRIES << "retries. Aborting receive.";

}

The code works but sending the byte of data seems to be the issue as it is recorded as 1’s all around resulting in bytes being 0xFF all around.



This is something that can hopefully be worked on by implementing signals for when sending one bit at a time.