Файл config.h:

#ifndef CONFIG\_H

#define CONFIG\_H

#define FRAME\_HEIGHT 720 // for transfer

#define FRAME\_WIDTH 1080 // for transfer

#define FPS 60 // fps

#define PACK\_SIZE 60176 // < max UDP packet size

#define ENCODE\_QUALITY 90 // larger=more quality but large packet sizes

#define IMAGE\_UDP\_PORT 50000

#define AUDIO\_UDP\_PORT 55455

#define IO\_TCP\_PORT 50001

#define MOUSEMOVE\_DELAY\_NS 10000000

#define SETTINGS\_FILE "config.ini"

#endif

Файл event.h:

#pragma once

#include <iostream>

#include <sys/types.h>

#include <sys/socket.h>

enum class Mode

{

KeyDown,

KeyUp,

MouseDown,

MouseUp,

MouseMove

};

inline std::string get\_mode\_value(const Mode &m)

{

switch (m)

{

case Mode::KeyDown:

return "KeyDown";

case Mode::KeyUp:

return "KeyUp";

case Mode::MouseDown:

return "MouseDown";

case Mode::MouseUp:

return "MouseUp";

case Mode::MouseMove:

return "MouseMove";

}

return "Unknown";

}

inline std::ostream &operator<<(std::ostream &out, const Mode value)

{

return out << get\_mode\_value(value);

}

struct Event

{

Mode mode;

int code;

int x;

int y;

};

inline std::ostream &operator<<(std::ostream &os, const Event &event)

{

return os << "Mode: " << event.mode << " Code: " << event.code << " X: " << event.x << " Y: " << event.y;

}

inline bool write(int socket, Event req)

{

auto bytes\_sent = ::send(socket, &req, sizeof(Event), 0);

return bytes\_sent == sizeof(Event);

}

inline bool read(int socket, Event \*buf)

{

auto bytes\_read = ::read(socket, buf, sizeof(Event));

return bytes\_read == sizeof(Event);

}

Файл grabber.cpp:

#include "utils.h"

#include "grabber.h"

#include "config.h"

#include <optional>

#include <sys/types.h>

#include <signal.h>

#include "popen2.h"

std::pair<int, int> get\_current\_mouse\_location()

{

auto output = exec("xdotool getmouselocation");

auto parts = Split(output, " ");

if (parts.size() < 2)

return {0, 0};

auto x = std::stoi(parts[0].substr(2));

auto y = std::stoi(parts[1].substr(2));

return {x, y};

}

std::optional<Mode> parse\_keyboard(const std::string &line, std::optional<Mode> mode, std::optional<Event> &event)

{

if (line.starts\_with(" detail: "))

{

auto num = std::stoi(line.substr(12));

event.emplace(Event{mode.value(), num, 0, 0});

return std::nullopt;

}

return mode;

}

std::optional<Mode> parse\_click(const std::string &line, std::optional<Mode> mode, std::optional<Event> &event)

{

if (line.starts\_with(" detail: "))

{

auto code = 0;

if (line.size() > 12)

code = std::stoll(line.substr(12));

auto coords = get\_current\_mouse\_location();

event.emplace(Event{mode.value(), code, coords.first, coords.second});

return std::nullopt;

}

return mode;

}

X11Grabber::~X11Grabber()

{

X11Grabber::stop();

}

void X11Grabber::start()

{

if (running.load())

return;

running.store(true);

thread = std::thread(&X11Grabber::loop, this);

}

void X11Grabber::stop()

{

if (!running.load())

return;

running.store(false);

thread.join();

}

void X11Grabber::set\_callback(std::function<void(const Event &)> callback)

{

this->callback = callback;

}

void X11Grabber::loop()

{

pid\_t child\_pid;

auto input = popen2("xinput test-xi2 --root", "r", &child\_pid);

if (!input)

{

fprintf(stderr,

"incorrect parameters or too many files.\n");

return;

}

std::optional<Mode> mode = std::nullopt;

while (running.load() && !std::feof(input))

{

char buffer[BUFFER\_SIZE] = {0};

if (std::fgets(buffer, BUFFER\_SIZE, input) != NULL)

{

std::string lines = buffer;

for (const auto &line : Split(lines, "\n"))

{

if (!mode.has\_value())

{

if (line.starts\_with("EVENT type 2"))

mode = Mode::KeyDown;

else if (line.starts\_with("EVENT type 3"))

mode = Mode::KeyUp;

else if (line.starts\_with("EVENT type 4"))

mode = Mode::MouseDown;

else if (line.starts\_with("EVENT type 5"))

mode = Mode::MouseUp;

else if (line.starts\_with("EVENT type 15"))

mode = Mode::MouseDown;

else if (line.starts\_with("EVENT type 16"))

mode = Mode::MouseUp;

else if (line.starts\_with("EVENT type 17"))

mode = Mode::MouseMove;

continue;

}

std::optional<Event> event = std::nullopt;

switch (mode.value())

{

case Mode::KeyDown:

mode = parse\_keyboard(line, Mode::KeyDown, event);

break;

case Mode::KeyUp:

mode = parse\_keyboard(line, Mode::KeyUp, event);

break;

case Mode::MouseDown:

mode = parse\_click(line, Mode::MouseDown, event);

break;

case Mode::MouseUp:

mode = parse\_click(line, Mode::MouseUp, event);

break;

case Mode::MouseMove:

mode = parse\_click(line, Mode::MouseMove, event);

break;

}

if (event.has\_value())

callback(event.value());

}

}

}

killpg(getpgid(child\_pid), SIGKILL);

pclose(input);

}

void GrabberSender::set\_host(std::string host)

{

this->host = host;

server\_address.sin\_family = AF\_INET;

server\_address.sin\_addr.s\_addr = inet\_addr(this->host.c\_str());

server\_address.sin\_port = htons(IO\_TCP\_PORT);

}

void GrabberSender::start()

{

socket = ::socket(AF\_INET, SOCK\_STREAM, 0);

if (socket == -1)

{

std::cerr << "Failed to create socket" << std::endl;

return;

}

if (::connect(socket, (struct sockaddr \*)&server\_address, sizeof(struct sockaddr\_in)) == -1)

{

::close(socket);

std::cerr << "Failed to connect to server" << std::endl;

return;

}

set\_callback([this](const Event &event)

{

if (event.mode == Mode::MouseMove) {

struct timespec now;

clock\_gettime(CLOCK\_MONOTONIC, &now);

if (now.tv\_sec - ts.tv\_sec < 1 && now.tv\_nsec - ts.tv\_nsec < MOUSEMOVE\_DELAY\_NS)

return;

ts = now;

}

write(socket, event); });

X11Grabber::start();

}

void GrabberSender::stop()

{

X11Grabber::stop();

char buf[2] = {0};

::send(socket, buf, sizeof(buf), 0); // termination command

::close(socket);

}

GrabberSender::~GrabberSender()

{

GrabberSender::stop();

}

Файл grabber.h:

#pragma once

#include <string>

#include <utility>

#include <optional>

#include <functional>

#include <thread>

#include <arpa/inet.h>

#include "event.h"

#define BUFFER\_SIZE 1024 \* 1024

std::pair<int, int> get\_current\_mouse\_location();

std::optional<Mode> parse\_keyboard(const std::string &line, std::optional<Mode> mode, std::optional<Event> &event);

std::optional<Mode> parse\_click(const std::string &line, std::optional<Mode> mode, std::optional<Event> &event);

class X11Grabber

{

public:

X11Grabber() = default;

virtual ~X11Grabber();

virtual void start();

virtual void stop();

void set\_callback(std::function<void(const Event &)> callback);

private:

std::function<void(const Event &)> callback;

std::thread thread;

std::atomic<bool> running = false;

void loop();

};

class GrabberSender : public X11Grabber

{

public:

GrabberSender() = default;

void set\_host(std::string host);

void start() override;

void stop() override;

~GrabberSender() override;

private:

std::string host;

struct timespec ts = {0, 0};

struct sockaddr\_in server\_address;

int socket;

};

Файл inputapplicant.cpp:

#include "inputapplicant.h"

#include "utils.h"

#include "config.h"

#include <sys/socket.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include <thread>

#include <cstring>

#include <signal.h>

X11InputApplicant::X11InputApplicant()

{

}

X11InputApplicant::~X11InputApplicant()

{

}

void X11InputApplicant::start()

{

if (running.test())

return;

running.test\_and\_set();

thread = std::thread(&X11InputApplicant::listen\_loop, this);

}

void X11InputApplicant::stop()

{

if (!running.test())

{

thread.join();

return;

}

running.clear();

::shutdown(listen\_socket, SHUT\_RDWR);

thread.join();

}

void X11InputApplicant::listen\_loop()

{

listen\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

int opt = 1;

if (setsockopt(listen\_socket, SOL\_SOCKET,

SO\_REUSEADDR, &opt, sizeof(opt)))

{

std::cerr << "Failed to create socket" << std::endl;

return;

}

if (listen\_socket == -1)

{

std::cerr << "Failed to create socket" << std::endl;

return;

}

struct sockaddr\_in server\_address;

server\_address.sin\_family = AF\_INET;

server\_address.sin\_addr.s\_addr = INADDR\_ANY;

server\_address.sin\_port = htons(IO\_TCP\_PORT);

if (bind(listen\_socket, (struct sockaddr \*)&server\_address, sizeof(server\_address)) == -1)

{

std::cerr << "Failed to bind socket" << std::endl;

::close(listen\_socket);

return;

}

if (listen(listen\_socket, 1) == -1)

{

std::cerr << "Failed to listen on socket" << std::endl;

::close(listen\_socket);

return;

}

std::cout << "Listening for incoming connections..." << std::endl;

struct sockaddr\_in client\_address;

socklen\_t client\_addr\_length = sizeof(client\_address);

int clientSocket = accept(listen\_socket, (struct sockaddr \*)&client\_address, &client\_addr\_length);

if (clientSocket == -1)

std::cerr << "Failed to accept connection" << std::endl;

while (running.test())

{

Event event;

if (!read(clientSocket, &event))

break;

consume(event);

}

::close(clientSocket);

::close(listen\_socket);

running.clear();

}

void X11InputApplicant::consume(const Event &event)

{

switch (event.mode)

{

case Mode::KeyDown:

exec("xdotool keydown " + std::to\_string(event.code));

break;

case Mode::KeyUp:

exec("xdotool keyup " + std::to\_string(event.code));

break;

case Mode::MouseDown:

exec("xdotool mousemove " + std::to\_string(event.x) + " " + std::to\_string(event.y) + " mousedown " + std::to\_string(event.code));

break;

case Mode::MouseUp:

exec("xdotool mousemove " + std::to\_string(event.x) + " " + std::to\_string(event.y) + " mouseup " + std::to\_string(event.code));

break;

case Mode::MouseMove:

exec("xdotool mousemove " + std::to\_string(event.x) + " " + std::to\_string(event.y));

break;

}

}

Файл inputapplicant.h:

#pragma once

#include <thread>

#include <atomic>

#include "event.h"

class X11InputApplicant

{

public:

X11InputApplicant();

~X11InputApplicant();

void start();

void listen\_loop();

void stop();

void consume(const Event &event);

private:

int listen\_socket;

std::thread thread;

std::atomic\_flag running = ATOMIC\_FLAG\_INIT;

};

Файл mainwindow.h:

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QPixmap>

#include <QAudioInput>

#include <QUdpSocket>

#include "zoomui.h"

#include "udpplayer.h"

class MainWindow : public QMainWindow, public Ui::MainWindow

{

Q\_OBJECT

private:

QPixmap mainimg;

QAudioInput \*audio\_input;

QUdpSocket \*audio\_socket;

bool mic\_enabled;

public:

MainWindow(QWidget \*parent = nullptr)

: QMainWindow(parent), mic\_enabled(true)

{

setupUi(this);

connect(micButton, SIGNAL(clicked()), this, SLOT(toggleMic()));

}

void init\_audio\_input(char \*server)

{

QAudioFormat format = getAudioFormat();

audio\_input = new QAudioInput(format);

audio\_socket = new QUdpSocket();

audio\_socket->connectToHost(server, AUDIO\_UDP\_PORT);

audio\_socket->waitForConnected();

start\_audio();

}

void start\_audio()

{

audio\_input->start(audio\_socket);

}

void stop\_audio()

{

audio\_input->stop();

}

void deinit\_audio\_input()

{

stop\_audio();

audio\_socket->deleteLater();

audio\_input->deleteLater();

}

public slots:

void processImage(const QImage &img)

{

imgpix = QPixmap::fromImage(img);

pixmap->setPixmap(imgpix);

}

void toggleMic()

{

mic\_enabled = !mic\_enabled;

QImage img(mic\_enabled == true ? ":/mic-on.png" : ":/mic-off.png");

if (mic\_enabled)

start\_audio();

else

stop\_audio();

micButton->setIcon(QPixmap::fromImage(img));

}

void beforeStopAll()

{

emit stopAll();

}

signals:

void stopAll();

};

#endif

Файл screenrecorder.cpp:

#include <string>

#include <vector>

#include <QApplication>

#include <QScreen>

#include <QDebug>

#include "opencv2/opencv.hpp"

#include <uvgrtp/lib.hh>

#include "screenrecorder.h"

#include "imageutil.h"

#include "cvmatandqimage.h"

#include "config.h"

using namespace cv;

void ScreenRecorder::run()

{

QRect rect = QApplication::screens().at(0)->geometry();

uvgrtp::context ctx;

uvgrtp::session \*sess = ctx.create\_session(server);

int flags = RCE\_FRAGMENT\_GENERIC | RCE\_SEND\_ONLY;

uvgrtp::media\_stream \*stream = sess->create\_stream(IMAGE\_UDP\_PORT, RTP\_FORMAT\_GENERIC, flags);

std::vector<uint8\_t> encoded;

uint32\_t frame\_counter = 0;

if (stream)

{

Mat image, send;

while (!QThread::currentThread()->isInterruptionRequested())

{

QPixmap pixmap = imageutil::takeScreenShot(rect);

image = QtOcv::image2Mat(pixmap.toImage());

resize(image, send, Size(FRAME\_WIDTH, FRAME\_HEIGHT), 0, 0, INTER\_LINEAR);

std::vector<int> compression\_params;

compression\_params.push\_back(IMWRITE\_JPEG\_QUALITY);

compression\_params.push\_back(quality);

imencode(".jpg", send, encoded, compression\_params);

int payload\_len = encoded.size();

int current\_seq = 0;

auto header\_frame = std::unique\_ptr<uint8\_t[]>(new uint8\_t[sizeof(uint32\_t) + 2 \* sizeof(int)]);

memcpy(header\_frame.get(), &frame\_counter, sizeof(frame\_counter));

memcpy(header\_frame.get() + sizeof(frame\_counter), &current\_seq, sizeof(current\_seq));

memcpy(header\_frame.get() + sizeof(frame\_counter) + sizeof(current\_seq), &payload\_len, sizeof(payload\_len));

stream->push\_frame(header\_frame.get(), sizeof(uint32\_t) + 2 \* sizeof(int), RTP\_NO\_FLAGS);

current\_seq++;

std::this\_thread::sleep\_for(std::chrono::milliseconds(frame\_interval));

int total\_pack = 1 + (payload\_len - 1) / pack\_size;

for (int i = 0; i < total\_pack; i++)

{

int to\_send = min<int>(pack\_size, payload\_len - i \* pack\_size);

auto frame = std::unique\_ptr<uint8\_t[]>(new uint8\_t[sizeof(uint32\_t) + sizeof(int) + to\_send]);

memcpy(frame.get(), &frame\_counter, sizeof(frame\_counter));

memcpy(frame.get() + sizeof(frame\_counter), &current\_seq, sizeof(current\_seq));

memcpy(frame.get() + sizeof(frame\_counter) + sizeof(current\_seq), encoded.data() + i \* pack\_size, to\_send);

stream->push\_frame(frame.get(), sizeof(uint32\_t) + sizeof(int) + to\_send, RTP\_NO\_FLAGS);

current\_seq++;

std::this\_thread::sleep\_for(std::chrono::milliseconds(frame\_interval));

}

frame\_counter++;

std::this\_thread::sleep\_for(std::chrono::milliseconds(frame\_interval));

}

sess->destroy\_stream(stream);

}

if (sess)

ctx.destroy\_session(sess);

}

Файл screenrecorder.h:

#ifndef SCREENRECORDER\_H

#define SCREENRECORDER\_H

#include <QThread>

#include <QMutex>

class ScreenRecorder : public QThread

{

Q\_OBJECT

private:

char \*server;

int pack\_size;

int frame\_interval;

int quality;

public:

ScreenRecorder(char \*server, int pack\_size, int frame\_interval, int quality)

: server(server), pack\_size(pack\_size), frame\_interval(frame\_interval), quality(quality) {}

protected:

virtual void run();

public slots:

void terminateThread()

{

if (isRunning())

{

requestInterruption();

wait();

}

}

};

#endif

Файл Server.cpp:

#include <iostream>

#include <cstdlib>

#include <map>

#include <set>

#include <QApplication>

#include <QMainWindow>

#include <QThread>

#include <QMutex>

#include <QDebug>

#include <QFile>

#include <QGraphicsPixmapItem>

#include <QMessageBox>

#include <QSettings>

#include <uvgrtp/lib.hh>

#include "opencv2/opencv.hpp"

using namespace cv;

#include "udpplayer.h"

#include "screenrecorder.h"

#include "config.h"

#include "zoomui.h"

#include "cvmatandqimage.h"

#include "workerthread.h"

#include "mainwindow.h"

#include "startwindow.h"

#include "settingswindow.h"

#include "grabber.h"

#include "inputapplicant.h"

constexpr int RECEIVER\_WAIT\_TIME\_MS = 10 \* 1000;

struct FrameData

{

int frame\_num;

int buffer\_size;

FrameData() : frame\_num(-1), buffer\_size(0) {}

FrameData(int frame\_num, int buffer\_size) : frame\_num(frame\_num), buffer\_size(buffer\_size) {}

};

struct FrameChunk

{

int seq;

int size;

uint8\_t \*data;

FrameChunk() : seq(-1), size(0), data(nullptr) {}

FrameChunk(int seq, int size, uint8\_t \*data) : seq(seq), size(size), data(data) {}

bool operator<(const FrameChunk &other) const

{

return seq < other.seq;

}

FrameChunk(const FrameChunk &other) : seq(other.seq), size(other.size)

{

data = new uint8\_t[size];

memcpy(data, other.data, size);

}

~FrameChunk()

{

delete[] data;

}

};

void MyThread::run()

{

uvgrtp::context ctx;

uvgrtp::session \*sess = ctx.create\_session("0.0.0.0");

int flags = RCE\_FRAGMENT\_GENERIC | RCE\_RECEIVE\_ONLY;

uvgrtp::media\_stream \*receiver = sess->create\_stream(IMAGE\_UDP\_PORT, RTP\_FORMAT\_GENERIC, flags);

if (receiver)

{

std::map<uint32\_t, FrameData> frames;

std::map<uint32\_t, std::set<FrameChunk>> chunks;

while (!QThread::currentThread()->isInterruptionRequested())

{

uvgrtp::frame::rtp\_frame \*frame = receiver->pull\_frame(RECEIVER\_WAIT\_TIME\_MS);

if (!frame)

break;

uint32\_t current\_frame;

int current\_seq;

memcpy(&current\_frame, frame->payload, sizeof(uint32\_t));

memcpy(&current\_seq, frame->payload + sizeof(uint32\_t), sizeof(int));

size\_t real\_len = frame->payload\_len - sizeof(uint32\_t) - sizeof(int);

uint8\_t \*data = new uint8\_t[real\_len];

memcpy(data, frame->payload + sizeof(uint32\_t) + sizeof(int), real\_len);

chunks[current\_frame].insert(FrameChunk(current\_seq, real\_len, data));

if (chunks[current\_frame].begin()->seq == 0) // received header

{

int buffer\_size;

memcpy(&buffer\_size, chunks[current\_frame].begin()->data, sizeof(int));

frames[current\_frame] = FrameData(current\_frame, buffer\_size);

chunks[current\_frame].erase(chunks[current\_frame].begin());

}

if (frames.count(current\_frame))

{

int offset = 0;

int buffer\_size = frames[current\_frame].buffer\_size;

uint8\_t \*buffer = new uint8\_t[buffer\_size];

for (auto it = chunks[current\_frame].begin(); it != chunks[current\_frame].end(); it++)

{

memcpy(buffer + offset, it->data, it->size);

offset += it->size;

}

if (offset == buffer\_size)

{

Mat rawData = Mat(1, buffer\_size, CV\_8UC1, buffer);

Mat cvimg = imdecode(rawData, IMREAD\_COLOR);

if (cvimg.size().width == 0)

{

std::cerr << "decode failure!" << std::endl;

continue;

}

resize(cvimg, cvimg, Size(1278, 638), 0, 0, INTER\_LINEAR);

QImage image = QtOcv::mat2Image(cvimg);

emit signalGUI(image);

frames.erase(current\_frame);

chunks.erase(current\_frame);

}

delete[] buffer;

}

(void)uvgrtp::frame::dealloc\_frame(frame);

}

sess->destroy\_stream(receiver);

}

if (sess)

ctx.destroy\_session(sess);

}

class SessionManager

{

private:

UDPPlayer \*player;

ScreenRecorder \*recorder;

MyThread \*listen\_thread;

MainWindow \*window;

StartWindow &startWindow;

SettingsWindow \*settingsWindow;

GrabberSender \*grabber;

X11InputApplicant \*inputApplicant;

std::string ConnectServer;

bool is\_control;

bool running = false;

public:

SessionManager(StartWindow &startWindow) : startWindow(startWindow)

{

QObject::connect(startWindow.settingsButton, &QPushButton::clicked, [&]()

{

settingsWindow = new SettingsWindow();

settingsWindow->setFixedSize(settingsWindow->width(), settingsWindow->height());

QObject::connect(settingsWindow->saveButton, &QPushButton::clicked, [&]()

{

settingsWindow->saveSettings();

settingsWindow->close();

settingsWindow->deleteLater(); });

settingsWindow->show(); });

}

void start()

{

if (running)

return;

running = true;

startWindow.hide();

player = new UDPPlayer();

QSettings settings(SETTINGS\_FILE, QSettings::IniFormat);

int pack\_size = settings.value("pack\_size", PACK\_SIZE).toInt();

int frame\_interval = (1000 / settings.value("fps", FPS).toInt());

int quality = settings.value("quality", ENCODE\_QUALITY).toInt();

if (!is\_control)

{

inputApplicant = new X11InputApplicant();

inputApplicant->start();

recorder = new ScreenRecorder((char \*)ConnectServer.c\_str(), pack\_size, frame\_interval, quality);

recorder->start();

QObject::connect(QApplication::instance(), SIGNAL(aboutToQuit()), recorder, SLOT(terminateThread()));

}

window = new MainWindow();

window->setFixedSize(window->width(), window->height());

QObject::connect(window->endButton, &QPushButton::clicked, [&](bool)

{

stop();

startWindow.show(); });

window->init\_audio\_input((char \*)ConnectServer.c\_str());

window->prepareScene(is\_control);

window->show();

if (is\_control)

{

grabber = new GrabberSender();

grabber->set\_host(ConnectServer);

grabber->start();

listen\_thread = new MyThread();

QObject::connect(listen\_thread, SIGNAL(signalGUI(const QImage &)), window, SLOT(processImage(const QImage &)));

QObject::connect(listen\_thread, &QThread::finished, window, &MainWindow::beforeStopAll);

QObject::connect(window, &MainWindow::stopAll, [&]()

{ stop();

startWindow.show(); });

listen\_thread->start();

QObject::connect(QApplication::instance(), SIGNAL(aboutToQuit()), listen\_thread, SLOT(terminateThread()));

}

}

void stop()

{

if (!running)

return;

running = false;

if (is\_control)

{

listen\_thread->terminateThread();

listen\_thread->deleteLater();

grabber->stop();

delete grabber;

grabber = nullptr;

}

else

{

recorder->terminateThread();

recorder->deleteLater();

inputApplicant->stop();

delete inputApplicant;

inputApplicant = nullptr;

}

window->deinit\_audio\_input();

player->deleteLater();

window->deleteLater();

}

void connectButtonClicked()

{

QString ip = startWindow.ipLabel->text();

if (ip.isEmpty() || QHostAddress(ip).isNull())

{

QMessageBox::warning(&startWindow, "Error", "Please enter an IP address");

return;

}

ConnectServer = ip.toLocal8Bit().data();

is\_control = startWindow.controlCheckbox->isChecked();

start();

}

};

int main(int argc, char \*\*argv)

{

QApplication app(argc, argv);

StartWindow startWindow;

startWindow.setFixedSize(startWindow.width(), startWindow.height());

SessionManager manager(startWindow);

QObject::connect(startWindow.connectButton, &QPushButton::clicked, [&]()

{ manager.connectButtonClicked(); });

startWindow.show();

return app.exec();

}

Файл settingswindow.h:

#ifndef UI\_SETTINGSWINDOW\_H

#define UI\_SETTINGSWINDOW\_H

#include <QtCore/QVariant>

#include <QtCore/QSettings>

#include <QtWidgets/QApplication>

#include <QtWidgets/QHBoxLayout>

#include <QtWidgets/QLabel>

#include <QtWidgets/QPushButton>

#include <QtWidgets/QSpacerItem>

#include <QtWidgets/QSpinBox>

#include <QtWidgets/QVBoxLayout>

#include <QtWidgets/QWidget>

#include <QtWidgets/QLineEdit>

#include "config.h"

class CustomSpinbox : public QSpinBox

{

Q\_OBJECT

public:

CustomSpinbox(QWidget \*parent = nullptr)

: QSpinBox(parent)

{

lineEdit()->setReadOnly(true);

}

};

class Ui\_SettingsWindow

{

public:

QLabel \*label;

QWidget \*verticalLayoutWidget;

QVBoxLayout \*verticalLayout;

QHBoxLayout \*horizontalLayout;

QLabel \*label\_2;

QSpinBox \*fpsVal;

QHBoxLayout \*horizontalLayout\_2;

QLabel \*label\_3;

CustomSpinbox \*packetVal;

QHBoxLayout \*horizontalLayout\_3;

QLabel \*label\_4;

QSpinBox \*qualityVal;

QWidget \*horizontalLayoutWidget\_4;

QHBoxLayout \*horizontalLayout\_4;

QSpacerItem \*horizontalSpacer;

QPushButton \*saveButton;

QSpacerItem \*horizontalSpacer\_2;

void setupUi(QWidget \*SettingsWindow)

{

if (SettingsWindow->objectName().isEmpty())

SettingsWindow->setObjectName(QString::fromUtf8("SettingsWindow"));

SettingsWindow->resize(800, 600);

label = new QLabel(SettingsWindow);

label->setObjectName(QString::fromUtf8("label"));

label->setGeometry(QRect(330, 20, 101, 41));

QFont font;

font.setPointSize(15);

font.setBold(true);

label->setFont(font);

label->setAlignment(Qt::AlignCenter);

verticalLayoutWidget = new QWidget(SettingsWindow);

verticalLayoutWidget->setObjectName(QString::fromUtf8("verticalLayoutWidget"));

verticalLayoutWidget->setGeometry(QRect(70, 160, 271, 241));

verticalLayout = new QVBoxLayout(verticalLayoutWidget);

verticalLayout->setObjectName(QString::fromUtf8("verticalLayout"));

verticalLayout->setContentsMargins(0, 0, 0, 0);

horizontalLayout = new QHBoxLayout();

horizontalLayout->setObjectName(QString::fromUtf8("horizontalLayout"));

label\_2 = new QLabel(verticalLayoutWidget);

label\_2->setObjectName(QString::fromUtf8("label\_2"));

QFont font1;

font1.setPointSize(12);

label\_2->setFont(font1);

horizontalLayout->addWidget(label\_2);

fpsVal = new QSpinBox(verticalLayoutWidget);

fpsVal->setObjectName(QString::fromUtf8("fpsVal"));

fpsVal->setMinimum(1);

fpsVal->setMaximum(60);

horizontalLayout->addWidget(fpsVal);

verticalLayout->addLayout(horizontalLayout);

horizontalLayout\_2 = new QHBoxLayout();

horizontalLayout\_2->setObjectName(QString::fromUtf8("horizontalLayout\_2"));

label\_3 = new QLabel(verticalLayoutWidget);

label\_3->setObjectName(QString::fromUtf8("label\_3"));

label\_3->setFont(font1);

horizontalLayout\_2->addWidget(label\_3);

packetVal = new CustomSpinbox(verticalLayoutWidget);

packetVal->setObjectName(QString::fromUtf8("packetVal"));

packetVal->setMinimum(1024);

packetVal->setMaximum(60416);

packetVal->setSingleStep(1024);

horizontalLayout\_2->addWidget(packetVal);

verticalLayout->addLayout(horizontalLayout\_2);

horizontalLayout\_3 = new QHBoxLayout();

horizontalLayout\_3->setObjectName(QString::fromUtf8("horizontalLayout\_3"));

label\_4 = new QLabel(verticalLayoutWidget);

label\_4->setObjectName(QString::fromUtf8("label\_4"));

label\_4->setFont(font1);

horizontalLayout\_3->addWidget(label\_4);

qualityVal = new QSpinBox(verticalLayoutWidget);

qualityVal->setObjectName(QString::fromUtf8("qualityVal"));

qualityVal->setMinimum(1);

qualityVal->setMaximum(100);

horizontalLayout\_3->addWidget(qualityVal);

verticalLayout->addLayout(horizontalLayout\_3);

horizontalLayoutWidget\_4 = new QWidget(SettingsWindow);

horizontalLayoutWidget\_4->setObjectName(QString::fromUtf8("horizontalLayoutWidget\_4"));

horizontalLayoutWidget\_4->setGeometry(QRect(-1, 510, 801, 80));

horizontalLayout\_4 = new QHBoxLayout(horizontalLayoutWidget\_4);

horizontalLayout\_4->setObjectName(QString::fromUtf8("horizontalLayout\_4"));

horizontalLayout\_4->setContentsMargins(0, 0, 0, 0);

horizontalSpacer = new QSpacerItem(40, 20, QSizePolicy::Expanding, QSizePolicy::Minimum);

horizontalLayout\_4->addItem(horizontalSpacer);

saveButton = new QPushButton(horizontalLayoutWidget\_4);

saveButton->setObjectName(QString::fromUtf8("saveButton"));

QSizePolicy sizePolicy(QSizePolicy::Minimum, QSizePolicy::Fixed);

sizePolicy.setHorizontalStretch(0);

sizePolicy.setVerticalStretch(0);

sizePolicy.setHeightForWidth(saveButton->sizePolicy().hasHeightForWidth());

saveButton->setSizePolicy(sizePolicy);

saveButton->setMinimumSize(QSize(200, 35));

horizontalLayout\_4->addWidget(saveButton);

horizontalSpacer\_2 = new QSpacerItem(40, 20, QSizePolicy::Expanding, QSizePolicy::Minimum);

horizontalLayout\_4->addItem(horizontalSpacer\_2);

retranslateUi(SettingsWindow);

QMetaObject::connectSlotsByName(SettingsWindow);

} // setupUi

void retranslateUi(QWidget \*SettingsWindow)

{

SettingsWindow->setWindowTitle(QCoreApplication::translate("SettingsWindow", "Settings", nullptr));

label->setText(QCoreApplication::translate("SettingsWindow", "Settings", nullptr));

label\_2->setText(QCoreApplication::translate("SettingsWindow", "FPS", nullptr));

label\_3->setText(QCoreApplication::translate("SettingsWindow", "Packet size", nullptr));

label\_4->setText(QCoreApplication::translate("SettingsWindow", "Quality", nullptr));

saveButton->setText(QCoreApplication::translate("SettingsWindow", "Save changes", nullptr));

} // retranslateUi

};

class SettingsWindow : public QWidget, public Ui\_SettingsWindow

{

Q\_OBJECT

public:

SettingsWindow(QWidget \*parent = nullptr)

: QWidget(parent)

{

setupUi(this);

QSettings settings(SETTINGS\_FILE, QSettings::IniFormat);

fpsVal->setValue(settings.value("fps", FPS).toInt());

packetVal->setValue(settings.value("packet", PACK\_SIZE).toInt());

qualityVal->setValue(settings.value("quality", ENCODE\_QUALITY).toInt());

}

void saveSettings()

{

QSettings settings(SETTINGS\_FILE, QSettings::IniFormat);

settings.setValue("fps", fpsVal->value());

settings.setValue("packet", packetVal->value());

settings.setValue("quality", qualityVal->value());

}

};

#endif

Файл startwindow.h:

#ifndef STARTWINDOW\_H

#define STARTWINDOW\_H

#include <QtCore/QVariant>

#include <QtGui/QIcon>

#include <QtWidgets/QApplication>

#include <QtWidgets/QCheckBox>

#include <QtWidgets/QLabel>

#include <QtWidgets/QLineEdit>

#include <QtWidgets/QPushButton>

#include <QtWidgets/QWidget>

class Ui\_StartWindow

{

public:

QLabel \*label;

QLabel \*label\_2;

QLineEdit \*ipLabel;

QPushButton \*connectButton;

QPushButton \*settingsButton;

QCheckBox \*controlCheckbox;

void setupUi(QWidget \*StartWindow)

{

if (StartWindow->objectName().isEmpty())

StartWindow->setObjectName(QString::fromUtf8("StartWindow"));

StartWindow->resize(1280, 720);

label = new QLabel(StartWindow);

label->setObjectName(QString::fromUtf8("label"));

label->setGeometry(QRect(510, 10, 241, 101));

QFont font;

font.setPointSize(15);

font.setBold(true);

label->setFont(font);

label->setTextFormat(Qt::AutoText);

label->setAlignment(Qt::AlignCenter);

label\_2 = new QLabel(StartWindow);

label\_2->setObjectName(QString::fromUtf8("label\_2"));

label\_2->setGeometry(QRect(510, 270, 251, 51));

QFont font1;

font1.setPointSize(15);

label\_2->setFont(font1);

label\_2->setAlignment(Qt::AlignCenter);

ipLabel = new QLineEdit(StartWindow);

ipLabel->setObjectName(QString::fromUtf8("ipLabel"));

ipLabel->setGeometry(QRect(520, 340, 231, 41));

ipLabel->setAlignment(Qt::AlignCenter);

connectButton = new QPushButton(StartWindow);

connectButton->setObjectName(QString::fromUtf8("connectButton"));

connectButton->setGeometry(QRect(600, 440, 80, 23));

connectButton->setStyleSheet(QString::fromUtf8("background-color: rgb(87, 227, 137);"));

settingsButton = new QPushButton(StartWindow);

settingsButton->setObjectName(QString::fromUtf8("settingsButton"));

settingsButton->setGeometry(QRect(1170, 640, 61, 51));

QIcon icon;

icon.addFile(QString::fromUtf8(":/config.png"), QSize(), QIcon::Normal, QIcon::Off);

settingsButton->setIcon(icon);

settingsButton->setIconSize(QSize(32, 32));

controlCheckbox = new QCheckBox(StartWindow);

controlCheckbox->setObjectName(QString::fromUtf8("controlCheckbox"));

controlCheckbox->setGeometry(QRect(570, 400, 141, 21));

retranslateUi(StartWindow);

QMetaObject::connectSlotsByName(StartWindow);

} // setupUi

void retranslateUi(QWidget \*StartWindow)

{

StartWindow->setWindowTitle(QCoreApplication::translate("StartWindow", "Remote control", nullptr));

label->setText(QCoreApplication::translate("StartWindow", "Remote control app", nullptr));

label\_2->setText(QCoreApplication::translate("StartWindow", "Connect to server:", nullptr));

ipLabel->setPlaceholderText(QCoreApplication::translate("StartWindow", "IP address", nullptr));

connectButton->setText(QCoreApplication::translate("StartWindow", "Connect", nullptr));

settingsButton->setText(QString());

controlCheckbox->setText(QCoreApplication::translate("StartWindow", "Control another PC", nullptr));

} // retranslateUi

};

class StartWindow : public QWidget, public Ui\_StartWindow

{

Q\_OBJECT

public:

StartWindow(QWidget \*parent = nullptr)

: QWidget(parent)

{

setupUi(this);

}

};

#endif

Файл udpplayer.cpp:

#include "udpplayer.h"

UDPPlayer::UDPPlayer(QObject \*parent) : QObject(parent)

{

socket = new QUdpSocket();

socket->bind(AUDIO\_UDP\_PORT);

QAudioFormat format = getAudioFormat();

QAudioDeviceInfo info(QAudioDeviceInfo::defaultOutputDevice());

if (!info.isFormatSupported(format))

format = info.nearestFormat(format);

output = new QAudioOutput(format);

device = output->start();

connect(socket, &QUdpSocket::readyRead, this, &UDPPlayer::playData);

}

void UDPPlayer::playData()

{

while (socket->hasPendingDatagrams())

{

QByteArray data;

data.resize(socket->pendingDatagramSize());

socket->readDatagram(data.data(), data.size());

device->write(data.data(), data.size());

}

}

QAudioFormat getAudioFormat()

{

QAudioFormat format;

format.setSampleRate(8000);

format.setChannelCount(1);

format.setSampleSize(16);

format.setByteOrder(QAudioFormat::LittleEndian);

format.setSampleType(QAudioFormat::SignedInt);

format.setCodec("audio/pcm");

QAudioDeviceInfo info(QAudioDeviceInfo::defaultInputDevice());

if (!info.isFormatSupported(format))

format = info.nearestFormat(format);

return format;

}

Файл udpplayer.h:

#ifndef UDPPLAYER\_H

#define UDPPLAYER\_H

#include <QObject>

#include <QtMultimedia/QAudioOutput>

#include <QtMultimedia/QAudioInput>

#include <QtMultimedia/QAudioFormat>

#include <QUdpSocket>

#include "config.h"

class UDPPlayer : public QObject

{

Q\_OBJECT

public:

explicit UDPPlayer(QObject \*parent = 0);

~UDPPlayer()

{

socket->deleteLater();

output->deleteLater();

}

private slots:

void playData();

private:

QAudioOutput \*output;

QUdpSocket \*socket;

QIODevice \*device;

};

QAudioFormat getAudioFormat();

#endif

Файл utils.cpp:

#include "utils.h"

#include <array>

#include <memory>

#include <stdexcept>

std::vector<std::string> Split(const std::string &string, const std::string &delimiter)

{

std::vector<std::string> result;

if (string.empty())

{

return result;

}

size\_t start = 0;

size\_t end = string.find(delimiter);

while (end != std::string::npos)

{

result.push\_back(string.substr(start, end - start));

start = end + delimiter.size();

end = string.find(delimiter, start);

}

result.push\_back(string.substr(start, end - start));

return result;

}

std::string exec(std::string cmd)

{

std::array<char, 128> buffer;

std::string result;

std::unique\_ptr<FILE, decltype(&pclose)> pipe(popen(cmd.c\_str(), "r"), pclose);

if (!pipe)

throw std::runtime\_error("popen() failed!");

while (fgets(buffer.data(), static\_cast<int>(buffer.size()), pipe.get()) != nullptr)

result += buffer.data();

return result;

}

Файл utils.h:

#pragma once

#include <string>

#include <vector>

std::vector<std::string> Split(const std::string &string, const std::string &delimiter = " ");

std::string exec(std::string cmd);

Файл workerthread.h:

#ifndef WORKERTHREAD\_H

#define WORKERTHREAD\_H

#include <QThread>

#include <QMutex>

#include <QImage>

class MyThread : public QThread

{

Q\_OBJECT

protected:

virtual void run();

signals:

void signalGUI(QImage);

public slots:

void terminateThread()

{

if (isRunning())

{

requestInterruption();

wait();

}

}

};

#endif

Файл zoomui.h:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Form generated from reading UI file 'zoomui.ui'

\*\*

\*\* Created by: Qt User Interface Compiler version 5.15.2

\*\*

\*\* WARNING! All changes made in this file will be lost when recompiling UI file!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef UI\_ZOOMUI\_H

#define UI\_ZOOMUI\_H

#include <QtCore/QVariant>

#include <QtGui/QIcon>

#include <QtWidgets/QApplication>

#include <QtWidgets/QFrame>

#include <QtWidgets/QGraphicsView>

#include <QtWidgets/QMainWindow>

#include <QtWidgets/QPushButton>

#include <QtWidgets/QWidget>

#include <QtWidgets/QLabel>

#include <QtWidgets/QGraphicsPixmapItem>

QT\_BEGIN\_NAMESPACE

class Ui\_MainWindow

{

public:

QWidget \*centralwidget;

QGraphicsScene \*graphicsScene;

QGraphicsView \*graphicsView;

QGraphicsPixmapItem \*pixmap;

QPixmap imgpix;

QFrame \*frame;

QPushButton \*micButton;

QPushButton \*endButton;

QPixmap watchImage;

void setupUi(QMainWindow \*MainWindow)

{

if (MainWindow->objectName().isEmpty())

MainWindow->setObjectName(QString::fromUtf8("MainWindow"));

MainWindow->resize(1280, 720);

centralwidget = new QWidget(MainWindow);

centralwidget->setObjectName(QString::fromUtf8("centralwidget"));

graphicsScene = new QGraphicsScene;

pixmap = new QGraphicsPixmapItem;

graphicsScene->addItem(pixmap);

graphicsView = new QGraphicsView(centralwidget);

graphicsView->setObjectName(QString::fromUtf8("graphicsView"));

graphicsView->setGeometry(QRect(0, 0, 1280, 640));

graphicsView->setScene(graphicsScene);

frame = new QFrame(centralwidget);

frame->setObjectName(QString::fromUtf8("frame"));

frame->setGeometry(QRect(0, 639, 1281, 102));

frame->setStyleSheet(QString::fromUtf8("background-color: gray"));

frame->setFrameShape(QFrame::StyledPanel);

frame->setFrameShadow(QFrame::Raised);

micButton = new QPushButton(frame);

micButton->setObjectName(QString::fromUtf8("pushButton"));

micButton->setGeometry(QRect(0, 0, 91, 81));

micButton->setAutoFillBackground(false);

micButton->setIcon(QPixmap::fromImage(QImage(":/mic-on.png")));

micButton->setIconSize(QSize(64, 64));

endButton = new QPushButton(frame);

endButton->setObjectName(QString::fromUtf8("endButton"));

endButton->setGeometry(QRect(1150, 20, 101, 41));

endButton->setStyleSheet(QString::fromUtf8("background-color: red;\n"

"color: rgb(255, 255, 255);"));

MainWindow->setCentralWidget(centralwidget);

watchImage = QPixmap::fromImage(QImage(":/control.png"));

retranslateUi(MainWindow);

QMetaObject::connectSlotsByName(MainWindow);

} // setupUi

void retranslateUi(QMainWindow \*MainWindow)

{

MainWindow->setWindowTitle(QCoreApplication::translate("MainWindow", "Remote control", nullptr));

micButton->setText(QString());

endButton->setText(QCoreApplication::translate("MainWindow", "End", nullptr));

} // retranslateUi

};

namespace Ui

{

class MainWindow : public Ui\_MainWindow

{

public:

void prepareScene(bool is\_control)

{

if (!is\_control)

pixmap->setPixmap(watchImage);

}

};

} // namespace Ui

QT\_END\_NAMESPACE

#endif // UI\_ZOOMUI\_H