Министерство образования Республики Беларусь
Учреждение Образования
«Брестский Государственный Технический Университет»
Кафедра ИИТ

Лабораторная работа №1-2
По дисциплине ОСиСП за 5 семестр
Тема: Разработка приложения: «Игра Lines»

Выполнил:
Студент 3-го курса
Группы ПО-5
Крощук В.В.
Проверила:
Дряпко А.В.

Брест 2021

Лабораторная работа №1-2

Вариант 11

Цель работы:
приобрести практические навыки проектирования и разработки приложений с графическим пользовательским интерфейсом в ОС Windows средствами Qt.

Задания и выполненные решения:

Игра «Lines». Реализовать игру по следующим правилам: имеется квадратное поле 10 X 10 клеток, в случайных ячейках которого в начале игры находятся пять цветных шариков (основные цвета - желтый, красный, зеленый).

На каждой итерации игрового процесса игрок может перетащить один произвольный шарик в любую позицию, достижимую из данной (т.е. из которой можно построить путь по свободным клеткам). После перетаскивания шарика на случайных незанятых позициях игрового поля снова появляются 5 шариков произвольного цвета.

Если игроку удается собрать цепочку из 4 шариков одного цвета, то они пропадают, а игроку начисляются очки. Игра ведется до полного заполнения игрового поля цветными шариками.

Цель игры – заработать максимальное количество очков.

Код программы:

main.cpp:

```
#include "mainwindow.h"
#include "Definitions.h"
#include <QtCore>
#include <QApplication>
#include <iostream>
#include "GameLogic.h"

using namespace std;

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.startScreen();
    return a.exec();
}
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

QImage createImageWithOverlay(const QImage& baseImage, const QImage& overlayImage);

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QPixmap bkgnd(":/Assets/background.png");
    QIcon icon(":/Assets/ghost_green.png");
    this->setWindowIcon(icon);
    QPalette palette;
```

```
    this->setPalette(palette);
    this->setFixedHeight(sWidth + 100);
    this->setFixedWidth(sWidth);
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question( this, "Exit?",
                                                                tr("Are you sure to
close the game?"),
                                                                QMessageBox::No |
QMessageBox::Yes,
                                                                QMessageBox::No);
    if (resBtn != QMessageBox::Yes) {
        event->ignore();
    } else {
        event->accept();
    }
}

MainWindow::~MainWindow()
{
    delete board;
    for(int i = 0; i != boardRow; ++i)
    {
        for(int j = 0; j != boardColumn; ++j)
        {
            delete[] cells[i][j];
        }
        delete[] cells[i];

    }
    delete[] cells;
    delete lastCellPlace;
}

void MainWindow::playButtonWasClicked(bool) {
    startGame();
}

void MainWindow::settingsButtonWasClicked(bool) {
    QMessageBox::StandardButton reply;
    reply = QMessageBox::question(this, "Exit?", "Are you sure to close the game?",
                                  QMessageBox::Yes|QMessageBox::No,
QMessageBox::No);
    if (reply == QMessageBox::Yes) {
        qDebug() << "Yes was clicked";
        QApplication::exit();
    } else {
        qDebug() << "Yes was *not* clicked";
    }

}

void MainWindow::homeButtonWasClicked(bool) {
    QMessageBox::StandardButton reply;
    reply = QMessageBox::question(this, "Go back", "You are going to cancel the
game. Are you sure?",
                                  QMessageBox::Yes|QMessageBox::No);
    if (reply == QMessageBox::Yes) {
        qDebug() << "Yes was clicked";
        startScreen();
    } else {
        qDebug() << "Yes was *not* clicked";
    }
}

void MainWindow::buttonWasPressed(QWidget* buttonW) {
    QPushButton* button = (QPushButton*)(buttonW);
```

```
    button->setIconSize(QSize(button->iconSize().width() / 2, button-
>iconSize().height() / 2));
}

void MainWindow::buttonWasReleased(QWidget* buttonW) {
    QPushButton* button = (QPushButton*)(buttonW);
    button->setIconSize(QSize(button->iconSize().width() * 2, button-
>iconSize().height() * 2));
}

void MainWindow::startScreen() {
    GameLogic::m_pInstance = NULL;
    GameLogic::window = this;

    int id = QFontDatabase::addApplicationFont("://Assets/ConcertOne-Regular.ttf");
    QString family = QFontDatabase::applicationFontFamilies(id).at(0);
    QFont monospace(family);
    monospace.setPointSize(20);

    QWidget *parent = new QWidget();
    parent->resize(sWidth, sWidth);

    QPushButton* playGame = new QPushButton(QIcon("://Assets/button_play.png"),
"Play", parent);
    playGame->move(100, 110);
    playGame->setFont(monospace);
    playGame->setStyleSheet("border: 0px; color: rgb(255, 255, 255)");
    playGame->setIconSize(QSize(200, 200));
    connect(playGame, SIGNAL(clicked(bool)), this,
SLOT(playButtonWasClicked(bool)));

    QPushButton* settings = new QPushButton(QIcon("://Assets/button_home.png"),
"Exit", parent);
    settings->move(200, 400);
    settings->setFont(monospace);
    settings->setStyleSheet("border: 0px; color: rgb(255, 255, 255)");
    settings->setIconSize(QSize(200, 200));

    connect(settings, SIGNAL(clicked(bool)), this,
SLOT(settingsButtonWasClicked(bool)));

    this->setCentralWidget(parent);
    this->show();
}

void MainWindow::startGame() {
    board = new QWidget();
    cells = new Cell* *[boardRow];
    for(int i = 0; i < boardRow; i++) {
        cells[i] = new Cell*[boardColumn];
    }
    QGridLayout* layout = new QGridLayout();

    this->setCentralWidget(board);
    board->setLayout(layout);
    for(int i = 0; i < boardRow; i++)
        for(int j = 0; j < boardColumn; j++){
            Cell* temp = new Cell();
            temp->place = MatrixPoint(i,j);
            QImage cell;
            (i + j) % 2 == 0 ? cell = QImage("://Assets/cell_light.png") : cell =
QImage("://Assets/cell_dark.png");
            temp->setIcon(QIcon(QPixmap::fromImage(cell)));
            temp->setStyleSheet("border: 0px");
            temp->setIconSize(QSize(cellSize, cellSize));
```

```
            connect(temp, SIGNAL(wasPressed(MatrixPoint)), GameLogic::Instance(),
SLOT(cellWasPressed(MatrixPoint)));
            connect(temp, SIGNAL(clicked(bool)), temp, SLOT(idiotClick(bool)));

            temp->setMinimumSize(cellSize, cellSize);
            layout->addWidget(temp, i, j);
            cells[i][j] = temp;
        }

    int id = QFontDatabase::addApplicationFont("://Assets/ConcertOne-Regular.ttf");
    QString family = QFontDatabase::applicationFontFamilies(id).at(0);
    QFont monospace(family);
    monospace.setPointSize(30);

    QPushButton* score = new QPushButton(QIcon("://Assets/icon_path.png"), "0",
this);
    score->setFont(monospace);
    score->setStyleSheet("border: 0px; color: rgb(255, 255, 255)");
    score->setIconSize(QSize(100, 100));

    layout->addWidget(score, boardRow + 1, 0, 2, 5);
    this->score = score;

    QPushButton* home = new QPushButton(QIcon("://Assets/button_home.png"), "",
this);
    home ->setFont(monospace);
    home->setStyleSheet("border: 0px; color: rgb(255, 255, 255)");
    home->setIconSize(QSize(100, 100));

    connect(home, SIGNAL(clicked(bool)), this, SLOT(homeButtonWasClicked(bool)));

    layout->addWidget(home, boardRow + 1, 5, 2, 2);

    GameLogic::Instance()->generateGhosts();

    this->show();
}

void MainWindow::ghostWasMoved(std::vector<Node> road, Ghosts type) {
    QSequentialAnimationGroup* animationManager = new QSequentialAnimationGroup();
    connect(animationManager, SIGNAL(finished()), this, SLOT(finishedAnimating()));
    lastCellPlace = new MatrixPoint(road.back().y, road.back().x);
    lastCellType = type;
    QImage cell;
    (road[0].y + road[0].x) % 2 == 0 ? cell = QImage("://Assets/cell_light.png") :
cell = QImage("://Assets/cell_dark.png");
    cells[road[0].y][road[0].x]->setIcon(QIcon(QPixmap::fromImage(cell)));
    for(unsigned int i = 1; i < road.size(); i++) {
        Node nextCell = road[i];
        Cell* current = cells[nextCell.y][nextCell.x];
        QPropertyAnimation *animation = new QPropertyAnimation(current,
"iconSize");
        animation->setDuration(50);
        animation->setStartValue(QSize(0, 0));
        animation->setEndValue(QSize(cellSize, cellSize));

        animationManager->addAnimation(animation);
    }
    animationManager->start();
}

void MainWindow::finishedAnimating() {
    if(lastCellPlace != NULL) {
        cells[lastCellPlace->row][lastCellPlace->column]-
>setIcon(mergedIcon(lastCellType, *lastCellPlace));
        lastCellPlace = NULL;
        GameLogic::Instance()->nextMove();
    }
```

```
}

QIcon MainWindow::mergedIcon(Ghosts type, MatrixPoint place, bool select) {
    QImage icon;
    switch (type) {
    case yellow:
        icon = QImage("://Assets/ghost_yellow.png");
        break;
    case white:
        icon = QImage("://Assets/ghost_white.png");
        break;
    case green:
        icon = QImage("://Assets/ghost_green.png");
        break;
    case red:
        icon = QImage("://Assets/ghost_red.png");
        break;
    }
    QImage cellIcon;
    if(select)
        cellIcon = (place.row + place.column) % 2 == 0 ?
QImage("://Assets/cell_light.png") : QImage("://Assets/cell_dark.png");
    else
        cellIcon = QImage("://Assets/cell_selected.png");
    QImage merged = createImageWithOverlay(cellIcon, icon);
    return QIcon(QPixmap::fromImage(merged));

}

void MainWindow::ghostWasGenerated(Ghosts type, MatrixPoint place){
    Cell* current = cells[place.row][place.column];

    current->setIcon(mergedIcon(type, place));

    QPropertyAnimation *animation = new
QPropertyAnimation(cells[place.row][place.column], "iconSize");
    animation->setDuration(100);
    animation->setStartValue(QSize(0, 0));
    animation->setEndValue(QSize(cellSize, cellSize));
    animation->start();
}

void MainWindow::ghostWasDeleted(MatrixPoint place) {
    Cell* temp = cells[place.row][place.column];
    if((place.row + place.column) % 2 == 0)
        temp->setIcon(QIcon("://Assets/cell_light.png"));
    else
        temp->setIcon(QIcon("://Assets/cell_dark.png"));
    temp->setIconSize(QSize(cellSize, cellSize));
}

void MainWindow::ghostWasSelected(Ghosts type, MatrixPoint place) {
    if(GameLogic::Instance()->gameBoard[place.row][place.column])
        cells[place.row][place.column]->setIcon(mergedIcon(type, place, false));
}

void MainWindow::ghostWasDeselected(Ghosts type, MatrixPoint place) {
    if(GameLogic::Instance()->gameBoard[place.row][place.column])
        cells[place.row][place.column]->setIcon(mergedIcon(type, place, true));
}

void MainWindow::gameOver() {
    GameLogic::m_pInstance  = NULL;
    QMessageBox::StandardButton reply;
    reply = QMessageBox::question(this, "Game Over", "You have lost. Do you want to
try again?",
                                  QMessageBox::Yes|QMessageBox::No);
    if (reply == QMessageBox::Yes) {
```

```
        startGame();
    } else {
        startScreen();
    }
}

QImage createImageWithOverlay(const QImage& baseImage, const QImage& overlayImage)
{
    QImage imageWithOverlay = QImage(overlayImage.size(),
QImage::Format_ARGB32_Premultiplied);
    QPainter painter(&imageWithOverlay);

    painter.setCompositionMode(QPainter::CompositionMode_Source);
    painter.fillRect(imageWithOverlay.rect(), Qt::transparent);

    painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
    painter.drawImage(0, 0, baseImage);

    painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
    painter.drawImage(0, 0, overlayImage);

    painter.end();

    return imageWithOverlay;
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "GameLogic.h"
#include <QGridLayout>
#include "cell.h"
#include "Definitions.h"
#include <QPainter>
#include <QAnimationDriver>
#include <QAnimationGroup>
#include <QGraphicsOpacityEffect>
#include <QButtonGroup>
#include <QFontDatabase>
#include <QMessageBox>
#include <QCloseEvent>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QWidget* board = new QWidget();
    Cell* **cells;
    Ghosts lastCellType;
    MatrixPoint* lastCellPlace;
    void closeEvent(QCloseEvent *bar);

public slots:
    void ghostWasSelected(Ghosts type, MatrixPoint place);
    void ghostWasDeselected(Ghosts type, MatrixPoint place);
    void ghostWasGenerated(Ghosts type, MatrixPoint place);
    void ghostWasMoved(std::vector<Node> road, Ghosts type);
    void ghostWasDeleted(MatrixPoint place);
    void finishedAnimating();

    void playButtonWasClicked(bool zrtik);
```

```
    void settingsButtonWasClicked(bool zrtik);
    void homeButtonWasClicked(bool zrtik);

    void buttonWasPressed(QWidget* button);
    void buttonWasReleased(QWidget* button);

signals:
//    wasPressed(QWidget* button);
//    wasReleased(QPushButton* button);

public:
    explicit MainWindow(QWidget *parent = 0);
    QPushButton* score;

    void gameOver();
    void startScreen();
    void startGame();
    void nextMove();
    QIcon mergedIcon(Ghosts type, MatrixPoint place, bool select = true);
    Ui::MainWindow *ui;
    ~MainWindow();
};
```

#endif // MAINWINDOW_H

GameLogic.cpp:

```
#include "GameLogic.h"

MainWindow* GameLogic::window = NULL;
GameLogic* GameLogic::m_pInstance = NULL;


bool isValid(int visited[][boardColumn], int row, int col)
{
    return (row >= 0) && (row < boardRow) && (col >= 0) && (col < boardColumn)
            && GameLogic::Instance()->gameBoard[row][col] == NULL &&
visited[row][col] == -1;
}

GameLogic::GameLogic()
{
    gameBoard = new Ghost**[boardRow];
    for(int i = 0; i < boardRow; i++) {
        gameBoard[i] = new Ghost*[boardColumn];
        for(int j = 0; j < boardColumn; j++)
        {
            gameBoard[i][j] = NULL;
            MatrixPoint temp(i,j);
            freeCells.push_back(temp);
        }
    }
}

GameLogic* GameLogic::Instance() {
    if(!m_pInstance)
        m_pInstance = new GameLogic();
    return m_pInstance;
}
GameLogic::~GameLogic() {
    for(int i = 0; i != boardRow; ++i)
    {
        for(int j = 0; j != boardColumn; ++j)
        {
            delete[] gameBoard[i][j];
        }
        delete[] gameBoard[i];
    }
```

```
    delete[] gameBoard;
}

void GameLogic::generateGhosts() {
    if(freeCells.size() > ghostsAtMove ){
        srand(time(0));
        std::vector<Ghost*> deletableGhosts;
        for(int i = 0; i < ghostsAtMove; i++){
            int cellNumber = rand() % freeCells.size();
            Ghosts type = Ghosts(rand() % ghostTypeCount);
            Ghost* temp = new Ghost();
            QObject::connect(temp, SIGNAL(wasSelected(Ghosts, MatrixPoint)),
window, SLOT(ghostWasSelected(Ghosts, MatrixPoint)));
            QObject::connect(temp, SIGNAL(wasDeselected(Ghosts, MatrixPoint)),
window, SLOT(ghostWasDeselected(Ghosts, MatrixPoint)));
            QObject::connect(temp, SIGNAL(wasCreated(Ghosts, MatrixPoint)), window,
SLOT(ghostWasGenerated(Ghosts, MatrixPoint)));
            QObject::connect(temp, SIGNAL(wasMoved(std::vector<Node>,Ghosts)),
window, SLOT(ghostWasMoved(std::vector<Node>,Ghosts)));
            QObject::connect(temp, SIGNAL(deleteYourselfAziz(MatrixPoint)), window,
SLOT(ghostWasDeleted(MatrixPoint)));
            temp->configure(type, freeCells[cellNumber]);
            gameBoard[freeCells[cellNumber].row][freeCells[cellNumber].column] =
temp;
            freeCells.erase(freeCells.begin() + cellNumber);
            std::vector<Ghost*> ghosts = findCombinations(temp->currentPlace,
type);
            for(unsigned int i = 0; i < ghosts.size(); i++)
            {
                bool wasFound = false;
                for(unsigned int j = 0; j < deletableGhosts.size(); j++)
                    if(deletableGhosts[j]->currentPlace.row == ghosts[i]-
>currentPlace.row
                            && deletableGhosts[j]->currentPlace.column ==
ghosts[i]->currentPlace.column)
                    {
                        wasFound = true;
                        break;
                    }

                if(!wasFound)
                    deletableGhosts.push_back(ghosts[i]);
            }
        }
        deleteGhosts(deletableGhosts);
    }
    else
        window->gameOver();
}

std::vector<Node> GameLogic::shortestRoad(MatrixPoint begin, MatrixPoint end) {
    int moveByRow[] = { -1, 0, 0, 1 };
    int moveByColumn[] = { 0, -1, 1, 0 };
    bool visited[boardRow][boardColumn];
    memset(visited, false, sizeof(visited));
    std::queue<Node> queue;
    visited[begin.row][begin.column] = true;
    Node initialNode({begin.column, begin.row, 0});
    queue.push(initialNode);
    int minimumDistance = INT_MAX;
    std::vector<Node> road; //here we take all the nodes we have visited
    road.push_back(initialNode);
    while(queue.size() != 0)
    {
        Node node = queue.front();
        queue.pop();
        int currentColumn = node.x;
        int currentRow = node.y;
```

```
        int distance = node.distance;
        if (currentRow == end.row && currentColumn == end.column)
        {
            minimumDistance = distance; //so if we have found the road we should
start cleaning our road array keeping cells which will be used for constructing
road
            for(unsigned int i = 0; i < road.size(); i++) //this cleans the
"headlike" cells :D
                if(road[i].distance >= minimumDistance && (road[i].x != end.column
|| road[i].y != end.row)){
                    road.erase(road.begin() + i);
                    i--;
                }
            Node head = node;
            while(head.x != begin.column || head.y != begin.row)
            {
                int tempDistance = head.distance;
                for(unsigned int i = 0; i < road.size(); i++){ //backtracking
                    bool isNeighbour = false;
                    if((road[i].y == head.y && road[i].x == (head.x + 1)) ||
(road[i].y == head.y && road[i].x == (head.x - 1)) || (road[i].y == (head.y - 1) &&
road[i].x == head.x) || (road[i].y == (head.y + 1) && road[i].x == head.x))
                        isNeighbour = true;
                    if(road[i].distance == (tempDistance - 1) && !isNeighbour){
                        road.erase(road.begin() + i);
                        i--;
                    }
                }
                for(unsigned int i = 0; i < road.size(); i++)
                    if(road[i].distance == tempDistance - 1){
                        head = road[i];
                        break;
                    }
            }
            break;
        }
        for (int k = 0; k < 4; k++) //checking for each move
        {
            bool freeCell = false;
            bool notVisitedCell = false;
            bool validCell = ((currentRow + moveByRow[k] >= 0) && (currentRow +
moveByRow[k] < boardRow) && (currentColumn + moveByColumn[k] >= 0) &&
(currentColumn + moveByColumn[k] < boardColumn));
            if(validCell)
            {
                freeCell = (gameBoard[currentRow + moveByRow[k]][currentColumn +
moveByColumn[k]] == NULL);
                notVisitedCell = (!visited[currentRow + moveByRow[k]][currentColumn
+ moveByColumn[k]]);

            }
            if (freeCell && validCell && notVisitedCell)
            {
                visited[currentRow + moveByRow[k]][currentColumn + moveByColumn[k]]
= true;
                Node temp = {currentColumn + moveByColumn[k], currentRow +
moveByRow[k], distance +1};
                queue.push(temp);
                road.push_back(temp);
            }
        }
    }
    if (minimumDistance != INT_MAX){
        qDebug() << "shortest road's length " << minimumDistance;
    }
    else{
        qDebug() << "unavailable" << minimumDistance;
        return std::vector<Node>();
```

```
    }

    return road;
}

void GameLogic::cellWasPressed(MatrixPoint point) {
    if(clickedGhost != NULL) {
        if(gameBoard[point.row][point.column] == NULL)
        {
            moveGhost(clickedGhost->currentPlace, point);
            clickedGhost = NULL;
        }
        else if(point.row == clickedGhost->currentPlace.row && point.column ==
clickedGhost->currentPlace.column){
            emit(clickedGhost->wasDeselected(clickedGhost->type, point));
            clickedGhost = NULL;
        }
        else
        {
            emit(clickedGhost->wasDeselected(clickedGhost->type, clickedGhost-
>currentPlace));
            clickedGhost = gameBoard[point.row][point.column];
            emit(clickedGhost->wasSelected(clickedGhost->type, point));
        }
    }
    else {
        if(gameBoard[point.row][point.column] != NULL) {
            clickedGhost = gameBoard[point.row][point.column];
            emit(clickedGhost->wasSelected(clickedGhost->type, point));
        }
    }
}

void GameLogic::moveGhost(MatrixPoint from, MatrixPoint to) {
    ghostDestination = new MatrixPoint(to.row, to.column);
    std::vector<Node> road = shortestRoad(from, to);
    if(!road.empty()) {
        clickedGhost->currentPlace = to;
        gameBoard[from.row][from.column] = NULL;
        gameBoard[to.row][to.column] = clickedGhost;
        freeCells.push_back(from);
        for(unsigned int i = 0; i < freeCells.size(); i++)
        {
            if (freeCells[i].row == to.row && freeCells[i].column == to.column){
                freeCells.erase(freeCells.begin() + i);
                break;
            }
        }
        int zrtik = INT_MAX;
        for(unsigned int i = 0; i < road.size(); i++) {
            if(road[i].distance != zrtik)
                zrtik = road[i].distance;
            else {
                road.erase(road.begin() + i);
                i--;
            }
        }
        emit(clickedGhost->wasMoved(road, clickedGhost->type));

    }
    else {
        cellWasPressed(from);
    }
}

void GameLogic::nextMove() {
    if(ghostDestination){
        MatrixPoint to = *ghostDestination;
```

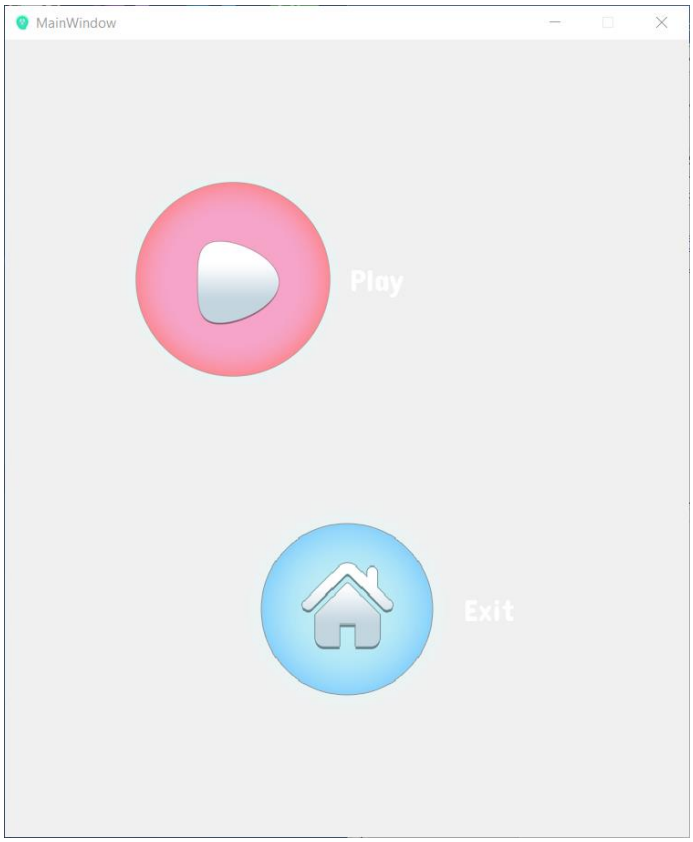
```
        std::vector<Ghost*> temp = findCombinations(to,
gameBoard[to.row][to.column]->type);
        deleteGhosts(temp);
        if(temp.size() == 0)
            generateGhosts();
    }
}

void GameLogic::deleteGhosts(std::vector<Ghost*> temp) {
    for(unsigned int i = 0; i < temp.size(); i ++) {
        emit(temp[i]->deleteYourselfAziz(temp[i]->currentPlace));
        gameBoard[temp[i]->currentPlace.row][temp[i]->currentPlace.column] = NULL;
        freeCells.push_back(temp[i]->currentPlace);
    }
    score += temp.size();
    window->score->setText(QString::number(score));
}

std::vector<Ghost*> GameLogic::findCombinations(MatrixPoint newlyAddedPlace, Ghosts
type) {
    std::vector<Ghost*> deletableGhosts;
    int rowRight = 0;
    for(int i = newlyAddedPlace.column + 1; i < boardColumn; i++){
        if(gameBoard[newlyAddedPlace.row][i] == NULL)
            break;
        if(gameBoard[newlyAddedPlace.row][i]->type == type)
            rowRight++;
        else
            break;
    }
    int rowLeft = 0;
    for(int i = newlyAddedPlace.column - 1; i >= 0 ; i--) {
        if(gameBoard[newlyAddedPlace.row][i] == NULL)
            break;
        if(gameBoard[newlyAddedPlace.row][i]->type == type)
            rowLeft++;
        else
            break;
    }
    int columnDown = 0;
    for(int i = newlyAddedPlace.row + 1; i < boardRow; i++) {
        if(gameBoard[i][newlyAddedPlace.column] == NULL)
            break;
        if(gameBoard[i][newlyAddedPlace.column]->type == type)
            columnDown++;
        else
            break;
    }
    int columnUp = 0;
    for(int i = newlyAddedPlace.row - 1; i >= 0; i--) {
        if(gameBoard[i][newlyAddedPlace.column] == NULL)
            break;
        if(gameBoard[i][newlyAddedPlace.column]->type == type)
            columnUp++;
        else
            break;
    }
    bool wasFound = false;
    if (columnDown + columnUp >= ghostBoomCount - 1) {
        for(int i = 0; i < columnUp; i++) {
            deletableGhosts.push_back(gameBoard[newlyAddedPlace.row - 1 -
i][newlyAddedPlace.column]);
        }
        for(int i = 0; i < columnDown; i++) {
            deletableGhosts.push_back(gameBoard[newlyAddedPlace.row + 1 +
i][newlyAddedPlace.column]);
        }
        wasFound = true;
```

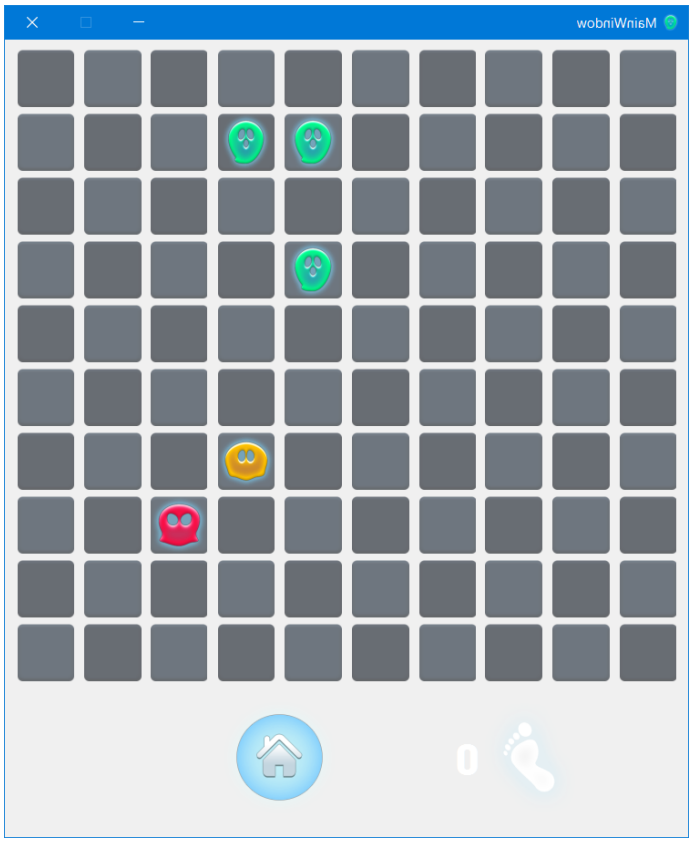
```
    }
    if (rowRight + rowLeft >= ghostBoomCount - 1) {
        for(int i = 0; i < rowRight; i++) {

deletableGhosts.push_back(gameBoard[newlyAddedPlace.row][newlyAddedPlace.column + 1
+ i]);
        }
        for(int i = 0; i < rowLeft; i++) {

deletableGhosts.push_back(gameBoard[newlyAddedPlace.row][newlyAddedPlace.column - 1
- i]);
        }
        wasFound = true;
    }
    if(wasFound)

deletableGhosts.push_back(gameBoard[newlyAddedPlace.row][newlyAddedPlace.column]);
    return deletableGhosts;
}
```

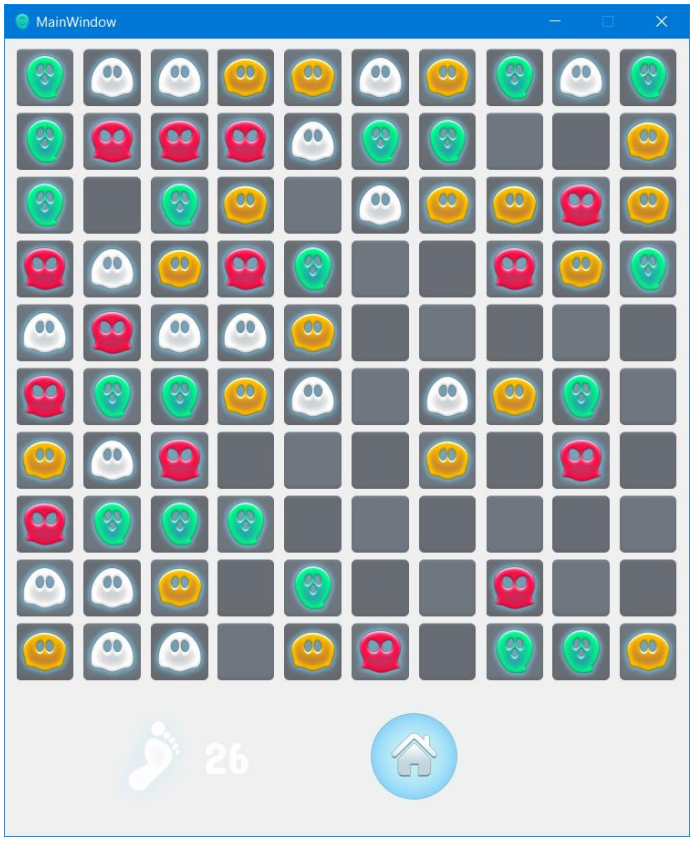
GameLogic.h:

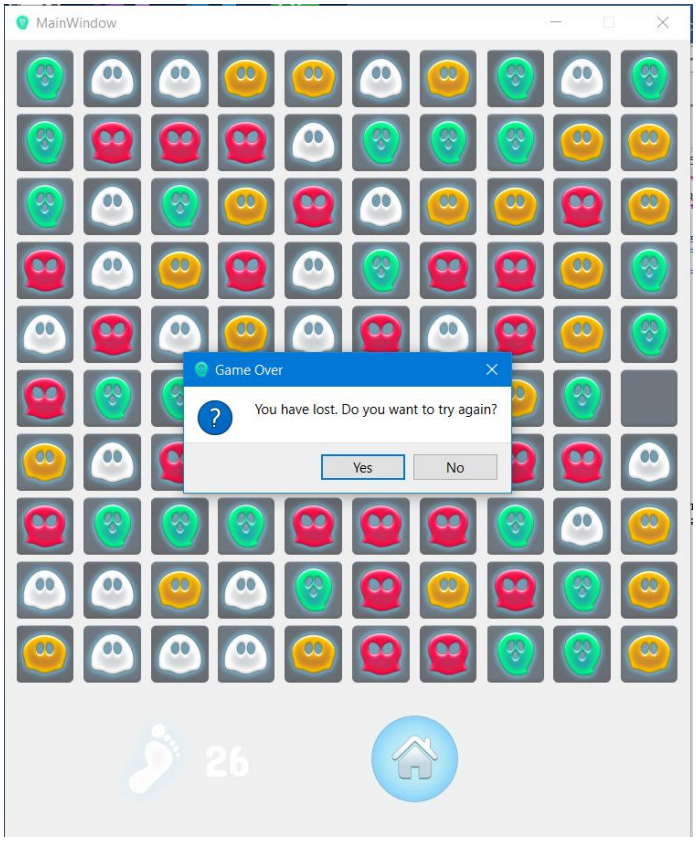
```
#ifndef GAMELOGIC_H
#define GAMELOGIC_H

#include "Ghost.h"
#include <iostream>
#include "Definitions.h"
#include "mainwindow.h"
#include <cstdlib>
#include <time.h>
#include <vector>
#include <QObject>
#include <queue>
#include <QDebug>

class MainWindow;

class GameLogic: public QObject
{
    Q_OBJECT
public:
    static GameLogic* m_pInstance;
    static MainWindow* window;
    Ghost* **gameBoard;
    std::vector<MatrixPoint> freeCells;
    MatrixPoint* ghostDestination = NULL;
    int score = 0;
    void nextMove();
private:
    Ghost* clickedGhost = NULL;
    GameLogic();
    ~GameLogic();
    void moveGhost(MatrixPoint, MatrixPoint);
    std::vector<Ghost*> findCombinations(MatrixPoint newlyAddedPlace, Ghosts type);
    std::vector<Node> shortestRoad(MatrixPoint from, MatrixPoint to);
    void deleteGhosts(std::vector<Ghost*> ghosts);

public:
    static GameLogic* Instance();
    void generateGhosts();
public slots:
    void cellWasPressed(MatrixPoint point);
};

#endif // GAMELOGIC_H
```

Работа программы:









Вывод: приобрел практические навыки проектирования и разработки приложений с графическим пользовательским интерфейсом в ОС Windows средствами Qt.