

Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проектированию

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему: «Разработка игрового агента для игры “Лабиринт”»

Выполнил:

Студент группы 23ВВВ3 Полиневский В. В.

Принял:

д. т. н. Митрохин М. А.

Пенза, 2024

**Содержание**

[Введение 5](#_Toc1)

[1. Постановка задачи 6](#_Toc2)

[2. Теоретическая часть задания 7](#_Toc3)

[3. Описание алгоритма программы 9](#_Toc4)

[4. Описание разработки программы 11](#_Toc5)

[4.1 Описание разработки визуальной составляющей проекта и внешней логики. 11](#_Toc6)

[4.1.1 Создание изображений. 12](#_Toc7)

[4.1.2 Объединение элементов в файл ресурсов. 13](#_Toc8)

[4.1.3. Описание QML компонентов: 14](#_Toc9)

[4.2Описание внутренней логики. 19](#_Toc10)

[Описание программы 30](#_Toc11)

[Заключение 32](#_Toc12)

[Список Литературы 33](#_Toc13)

[Приложение А.Листинг программы 34](#_Toc14)

# Введение

Лабиринт (англ. maze) — жанр [компьютерных игр](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B8%D0%B3%D1%80%D0%B0) игры данного жанра характеризуются тем, что успех игрока определяется в основном навигацией и ориентацией в лабиринте. Ответ на вопрос, является ли что-то лабиринтом, относителен, но как правило имеется возможность различить, ставит ли предлагаемая комбинация комнат и коридоров необходимость решения задачи навигации в лабиринте.

Игровой агент — это объект, который следит за своим окружением, принимает решения на основании этой информации и действует в соответствии с этими решениями. В случае для лабиринта игровой агент представляет собой алгоритм по достижению точки выхода из него.

Прохождение лабиринта — это задача нахождения пути из начальной точки (входа) в конечную точку (выход) в структуре, состоящей из множества путей и "стен" (преград).

В качестве среды для разработки мною была выбрана среда разработки Qt Creator 13.0.0, функционирующая на базе фраймворка используемый язык программирования – Си++, язык разметки - QML, система сборки - Cmake; используемый компилятор: GCC 13.2.0.

Целью данной курсовой работы является разработка графической программы, отражающей прохождение лабиринта. Основные языки программирования Си++ и QML. Именно с их помощью в данном курсовом проекте реализуется логика прохождения лабиринта игроком и игровым агентом и их графическое отображение.

# Постановка задачи

Требуется разработать программу, представляющую собой игру «Лабиринт», а также графическое оформление для вывода этого лабиринта в окно программы. Лабиринт будет проходиться пользователем и одновременно игровым агентом.

Исходный лабиринт храниться в виде матрицы, в которой будут заданы различные элементы (стена, пустое пространство, точка начала и конца, макет игрока и игрового агента). Размерность матрицы постоянна и не может быть задана игроком. Программа должна работать так, чтобы пользователь мог выбрать направление движения макета игрока при помощи клавиш стрелок на клавиатуре. Программа должна содержать простой и понятный графический интерфейс, определяющий необходимые действия, такие как генерация и выход из программы.

Устройства ввода – клавиатура и мышь.

# Описание алгоритма программы

Лабиринт представляет собой набор Различных элементов: стена, как преграда, точка старта и точка конца (выхода из лабиринта), пустое пустое пространство, доступное для передвижения.

Существует много различных классификаций лабиринтов: по размерности (2д и 3д), по гиперразмерности (не-гиперлабиринты, гиперлабиринт) и другие типы разделений, а также не меньшее количество способов их сгенерировать и пройти.

Моя курсовая работа посвящена прохождению классического лабиринта, в 2 д измерении с видом сверху. (Рисунок 1)

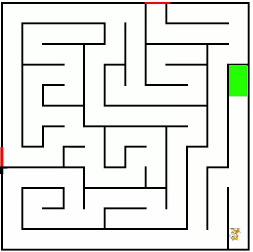


Рисунок 1 - Изображение лабиринта

Для генерации лабиринта я буду использовать алгоритм «Growing Tree».

Алгоритм Growing Tree — это один из методов генерации лабиринтов, который является универсальным и гибким. Он сочетает в себе элементы других алгоритмов (например, случайного блуждания и алгоритма поиска в глубину), позволяя создавать лабиринты с разной структурой. Основная идея алгоритма заключается в том, чтобы итеративно "растить" лабиринт, начиная с одной клетки и постепенно добавляя к нему новые пути.

Шаги алгоритма Growing Tree:

* Начальная инициализация:
* Создание сетку (грид) из клеток. Каждая клетка вначале считается "заполненной" или "стеной" (непроходимой);
* Выбор случайной начальной клетки и добавление её в "список активных клеток";
* Обозначение этой клетку как "посещённая";

Повторение итераций: Пока есть активные клетки в списке:

* Выбор клетки из списка активных клеток;
* Помещение как посещенной Крайней добавленной клетки (LIFO) — получится структура, похожая на DFS;
* Нахождение соседей выбранной клетки (Соседи — это клетки вокруг текущей клетки (например, сверху, снизу, слева, справа) которые ещё не посещены);
* Проверка что сосед находится в пределах границ сетки;

Если есть непосещённые соседи:

* Выборка одного из соседей случайным образом;
* Удаление стену между текущей клеткой и выбранным соседом ;
* Добавьте выбранного соседа в список активных клеток.
* Установка соседа как "посещённого";

Если непосещённых соседей нет:

* Удаление текущей клетки из списка активных клеток;
* Повторение процесса до тех пор, пока список активных клеток не станет пустым.

Для прохождения лабиринта игровым агентом я буду использовать алгоритм получения точного пути при помощи поиска в ширину, который описан в методе *findPath().*

Шаги алгоритма поиска в ширину для лабиринта:

* Получение размера лабиринта;
* Инициализация структуры с координатами начала и конца;
* Инициализация очереди для проходимого пути;

Цикл BFS:

Пока очередь не пуста:

* Извлекаем из очереди текучую точку;
* Проверка на достижение конца;
* Для текущей точки пытаемся перейти в соседние клетки по очереди (вправо, вниз, влево, вверх);
* Вычисление координат соседней клетки;
* Проверка клетки на доступность;
* При доступном переходе отмечаем как посещенную, создаем новый путь, дабавляем координаты в очередь.

Для программной реализации алгоритма в классе языка си++ понадобиться двумерные векторы My\_Maze и Game\_agent\_Maze (int), в котором будет находиться набор цифр: 0, 1, 2, 3, 4. Цифра 0 будет обозначать пустое пространство, 1 – стену, 2 – точку начала, 3 – точку конца, 4 – Игровую модель. Два вектора типа «QString»: myGameScreenImageData , agentScreenImageData; которые необходимы для хранения в них адресов изображений, используемых для создания изображения лабиринта.

# Описание разработки программы

Проект был создан при помощи средств IDE QT Qt Quick Application с системой сборки Сmake.

Выбор пал именно на эту IDE, так как из под коробки для используемой мной системы linux ubuntu, это самый оптимальный вариант из-за относительно простой установки необходимых программных компонентов и расширений. Также используемая мной версия Фраймворка QT 6.2 поддерживает функцию разработки интерфейса при помощи стандарта QML (Qt Modeling Language) основанный на Java Script. По синтаксису и решаемым задачам он походит на xml разметку.

Программа является многомодульной. Основные пользовательские модули и модули предусмотренные системой разработки: файл ресурсов res.qrc, Sourse Files: Logic\_of\_the\_maze.cpp и main.cpp, Header Files: Logic\_of\_the\_maze.h, файлы qml разметки main.

### **4.1 Описание разработки визуальной составляющей проекта и внешней логики.**

Так как для максимального достижения моих целей и задач важны визуальные результаты наличие GUI (Graphical User Interface) необходимо, так как он помогает пользователю видеть результат выполнения приложения в реальном времени, а также легко находить нужные для корректного выполнения программы элементы.

Первым этапом разработки стало создание графики для моей курсовой работы. Она представляет собой совокупность созданных изображений и компонентов QML. В приложении отсутствуют анимации, по причине ненадобности. Но при этом присутствует красочная палитра цветов. Скриншот главного экрана расположен ниже (Рисунок 2)

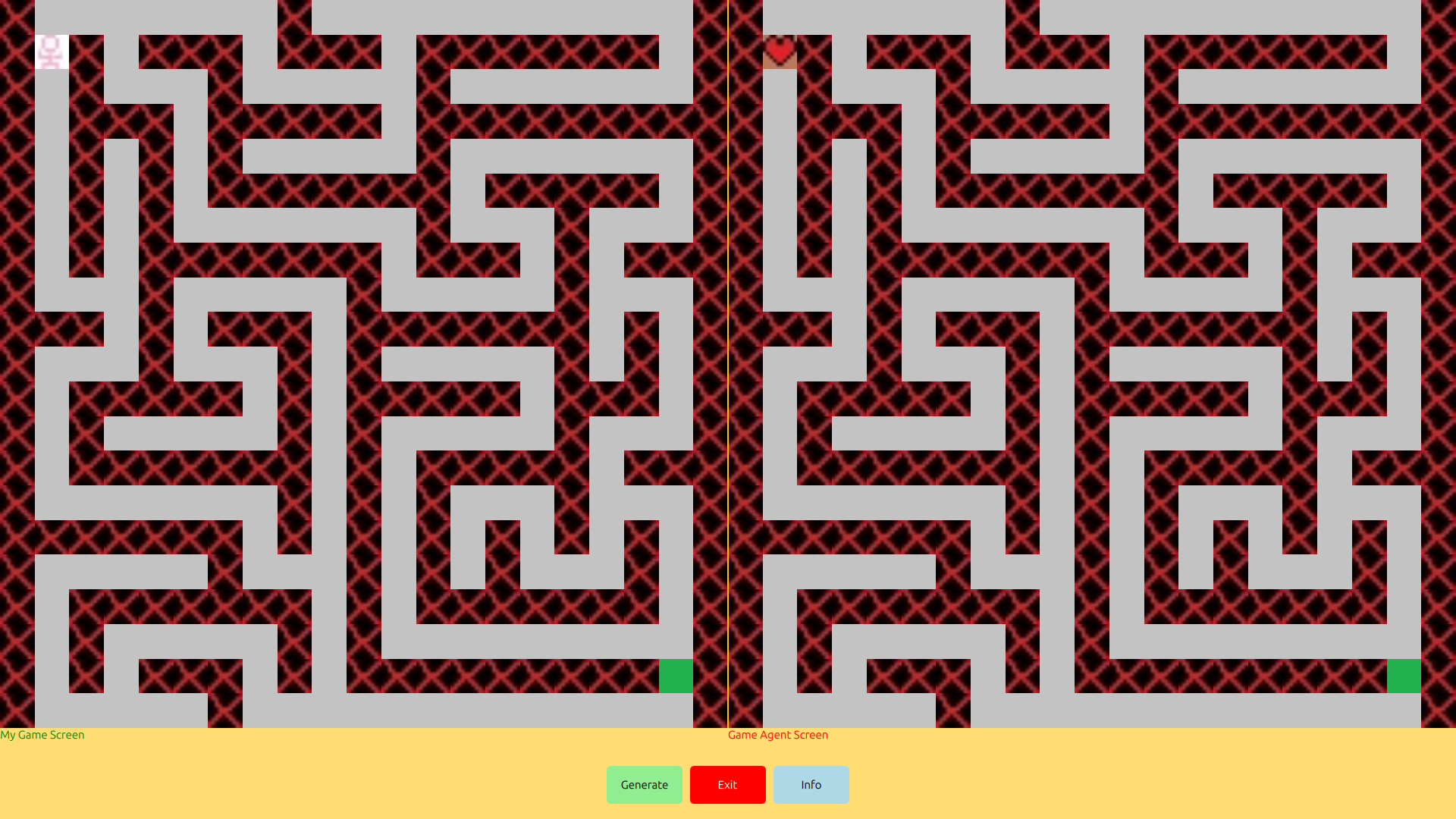


Рисунок 2 – Графический интерфейс программы

##### 4.1.1 Создание изображений.

Для отображения визуальных составляющих лабиринта, стен, пустого пространства, игровых моделей игрока и игрового агента и «точки конца» мной через графический редактор Paint были созданы квадратные изображения, имеющие размер 10 на 10 пикселей,

(Рисунок 3, Рисунок 4, Рисунок 5, Рисунок 6, Рисунок 7). Которые вы можете увидеть ниже.



Рисунок 3 – end\_element.jpg



Рисунок 4 – game\_agent\_element.jpg



Рисунок 5 – my\_game\_element.jpg



Рисунок 6 – void\_img.jpg



Рисунок 7 – well\_elemet.jpg

##### 4.1.2 Объединение элементов в файл ресурсов.

Файл ресурсов (Resource Collection Files, .qrc) — это файл, в котором указываются ресурсы, связанные с приложением.

В моем проекте файлом ресурсов является модуль res.qrc. В нем указаны файлы,расположенной внутри проекта: изображения из папки images, а также qml файлы, помимо main.qml, из папки qml\_others.

Res.qrc:

<RCC>

<qresource prefix="/qt/qml/Main">

<file>images/game\_agent\_element.jpg</file>

<file>images/my\_game\_element.jpg</file>

<file>images/void\_img.jpg</file>

<file>images/well\_element.jpg</file>

<file>images/end\_element.jpg</file>

<file>qml\_others/Information.qml</file>

<file>qml\_others/Game\_agent\_win.qml</file>

<file>qml\_others/Me\_win.qml</file>

<file>qml\_others/Information.qml</file>

</qresource>

</RCC>

##### 4.1.3. Описание **QML** компонентов**:**

**Модуль main.qml:**

Window {

Connections {

...

function onGameWon() {

...

}

function onAgentWon() {

...

}

}

Loader {

id: my\_Win\_Loader

source: "qml\_others/Me\_win.qml"

active: false

}

Loader {

id: myWindowLoader

source: "qml\_others/Information.qml"

active: false

}

Loader {

id: game\_Agent\_win

source: "qml\_others/Game\_agent\_win.qml"

active: false

}

...

Rectangle {

id: main\_Gaym\_Screen

...

Keys.onPressed: {

switch (event.key) {

...

}

}

function movePlayer(dx, dy) {

mazeData.movePlayer(dx, dy);

}

...

Rectangle{

id: my\_Game\_Screen

...

GridView {

id: myGameScreenGridView

...

}

}

...

Rectangle{

id: game\_Agent\_Screen

...

}

GridView {

id: game\_Agent\_Screen\_Grid\_View

...

}

...

}

}

}

RowLayout {

...

Rectangle {

id: generateButton

...

Text {

text: "Generate"

anchors.centerIn: parent

color: "black"

}

MouseArea {

anchors.fill: parent

onClicked: {

mazeData.logic\_generate\_maze();

//mazeData.fillScreenImages();

myGameScreenGridView.visible = true

game\_Agent\_Screen\_Grid\_View.visible = true

main\_Gaym\_Screen.focus = true

}

}

}

Rectangle {

id: exitButton

....

Text {

text: "Exit"

....

}

MouseArea {

anchors.fill: parent

onClicked: {

Qt.quit()

}

}

}

Rectangle {

id: infoButton

...

Text {

text: "Info"

....

}

MouseArea {

...

onClicked: {

myWindowLoader.active = true

}

}

}

}

}

Элемент Connection и вложенные в него Java Script Функции: function onGameWon() и function onAgentWon() , ответственны за логику получения и отображения результатов игры, победа и поражение.

В модуле задано основное окно Window, в котором расоложенны все функции Java Script, элементы разметки, известные как Встроенные QML модули: Rectangle , Row\_Layout и другие, а также прочее компоненты языка QML.

Элементы Loader ответственны за вызов всплывающих окон, расположенные в модулях: Me\_win.qml, Information.qml, Game\_agent\_win.qml. Контент на них отражает информацию, характерную для действий «Получение информации о программе», «Информирование о победе», «Информирование о проигрыше» соответственно.

Rectangle c id: «main\_Gaym\_Screen», является объектом с вложенной в себя функцией обработки нажатия клавиш-стелок на клавиатуре, function movePlayer(dx, dy). Также данный прямоугольник является родительским элементом для игровых полей, Rectangle: my\_Game\_Screen и game\_Agent\_Screen. Оба они также включают в себя GridView с описанием заполнения «Себя» изображениями лабиринта, такими как стена, пустота и другие.

RowLayout представляет собой элемент с возможностью расположения элементов внутри себя «side by size» (бок о бок). Он является родительским для кнопок, вернее, их имитацией элементами Rectangle c id: generateButton, exitButton и infoButton. Rectangle с id generateButton отвечает за вызов логики генерации, Rectangle c id exitButton – за вызов логики закрытия программы, Rectangle с id: infoButton – за вызов логики демонстрации о программе.

### 4.2 Описание внутренней логики.

Внутренняя логика описана в нескольких файлах: main.cpp, logic\_of\_the\_maze.cpp, logic\_of\_the\_maze.h.

**Logic\_of\_the\_maze.h:**

В данном заголовочном файле-классе описаны методы, для осуществления работы с данными, и контентом. Весь код модуля будет размещен в приложении А, ниже описаны основные алгоритмы и переменные класса и подключенные библиотеки.

Подключенные библиотеки:

#include <QTimer>

#include <QObject>

#include <QString>

#include <QVector>

#include <ctime>

#include <vector>

#include <cstdlib>

...

class Logic\_of\_the\_maze : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QVector<QStringList> myGameScreenImages READ getMyGameScreenImages NOTIFY myGameScreenImagesChanged)

Q\_PROPERTY(QVector<QStringList> agentScreenImages READ getAgentScreenImages NOTIFY agentScreenImagesChanged)

....

Задание Объекта, требуемого в QT и свойств класса для Qlist с изображениями лабиринта.

public:

using Maze = vector<vector<int>>;

Maze myMaze;

Maze Game\_agent\_Maze;

// Пути к изображениям

vector<vector<QString>> myGameScreenImageData;

vector<vector<QString>> agentScreenImageData;

QVector<QStringList> getMyGameScreenImages() const {

....

}

QVector<QStringList> getAgentScreenImages() const {

....

}

Опеределеление лабиринтов MyMaze и Maze\_Game\_agent\_Maze в виде двойного вектора тип int для хранения матриц лабиринтов.

agentScreenImageData, и myGameScreenImageData – Векторы из строк. Используются для хранения и обновления изображений в лабиринте.

getAgentScreenImages() и getAgentScreenImages() – представляют собой функции для получения размеров описанных выше массивов.

Private:

bool agentStarted = false;

QTimer\* agentTimer;

int currentPathIndex = 0;

В данном спецификаторе доступа описаны переменные agentStarted для логики начала прохождения ИА, \* agentTimer для отслеживания скорости выполнения обхода ИА, currentPathIndex – переменная для хранения текущей позиции в пути.

Public clots содержит несколько функций: void logic\_generate\_maze() c логикой генерации матриц лабиринта, алгоритм находится в описании решения. Вариация ниже void logic\_generate\_maze() {

// Устанавливаем новое "зерно" для генератора случайных чисел

static bool seeded = false;

if (!seeded) {

srand(static\_cast<unsigned int>(time(nullptr)));

seeded = true;

}

const int width = 21, height = 21;

myMaze.resize(height, vector<int>(width, 1));

// Выбираем случайную начальную точку

int startX = rand() % width;

int startY = rand() % height;

myMaze[startY][startX] = 0; // Начальная точка - путь

std::vector<std::pair<int, int>> stack;

stack.push\_back({startX, startY});

while (!stack.empty()) {

int x = stack.back().first;

int y = stack.back().second;

// Соседних непосещенные клетки

std::vector<std::pair<int, int>> neighbors;

if (x > 1 && myMaze[y][x - 2] == 1) neighbors.push\_back({x - 2, y});

if (x < width - 2 && myMaze[y][x + 2] == 1) neighbors.push\_back({x + 2, y});

if (y > 1 && myMaze[y - 2][x] == 1) neighbors.push\_back({x, y - 2});

if (y < height - 2 && myMaze[y + 2][x] == 1) neighbors.push\_back({x, y + 2});

if (!neighbors.empty()) {

// Выбираем случайного соседа и удаляем стену между ним и текущей клеткой

std::pair<int, int> next = neighbors[rand() % neighbors.size()];

myMaze[next.second][next.first] = 0;

myMaze[(next.second + y) / 2][(next.first + x) / 2] = 0;

stack.push\_back(next);

} else {

// Если нет непосещенных соседей, возвращаемся назад

stack.pop\_back();

}

}

// Устанавливаем агента и цель в лабиринте

myMaze[1][1] = 2; // Агент

myMaze[height - 2][width - 2] = 3; // Цель

Game\_agent\_Maze.resize(height , vector<int>(width, 1));

for (int i = 0; i < 21; i++){

for (int j = 0; j < 21; j++){

Game\_agent\_Maze[i][j] = myMaze[i][j];

}

}

fillScreenImages();

}

Void fillScreenImages() – функция, необходимая для инициализации и обновления матриц изображений для отображения на экран, работает при помощи switch и for. Структура ниже.

Структура:

For () {

For(){

Swith();

Swith();

}}

Данная функция также содержит в себе определение сигналов

emit myGameScreenImagesChanged();

emit agentScreenImagesChanged(); которые необходимы для qml метода GridView в main.qml

Q\_INVOKABLE void movePlayer(int dx, int dy) {

// Старт агента если первое движение

if (!agentStarted) {

startAgent();

}

// Находим текущую позицию игрока

auto [x, y] = findPlayerPosition(myMaze);

if (x == -1 || y == -1) {

return;

}

// Вычисляем новую позицию

int newX = x + dx;

int newY = y + dy;

// Проверяем границы лабиринта

if (newX < 0 || newX >= myMaze[0].size() || newY < 0 || newY >= myMaze.size()) {

return;

}

// Проверяем, не является ли новая позиция стеной

if (myMaze[newY][newX] == 1) {

return;

}

// Сохраняем значение целевой клетки

bool reachedGoal = (myMaze[newY][newX] == 3);

// Обновляем позицию игрока

myMaze[y][x] = 0; // Убираем игрока с текущей позиции

myMaze[newY][newX] = 2; // Устанавливаем игрока на новую позицию

// Обновляем отображение лабиринта

fillScreenImages();

// Проверяем достижение цели

if (reachedGoal) {

// Здесь можно добавить сигнал для оповещения о победе

emit gameWon(); // Добавьте этот сигнал в секцию signals

}

}

Метод void movePlayer(int dx, int dy) определяет передвижение модели игрового агента и перезапись изображений лабиринта при вызове функции fillScreenImages();

void moveAgent() {

if (!agentStarted) return;

auto [currentX, currentY] = findPlayerPosition(Game\_agent\_Maze);

if (currentX == -1 || currentY == -1) return;

// Если путь пуст или нужно его пересчитать

if (path.empty()) {

path = findPath(Game\_agent\_Maze);

if (path.empty()) return;

currentPathIndex = 1; // Пропускаем текущую позицию

}

// Проверяем, не достигли ли конца пути

if (currentPathIndex >= path.size()) {

agentStarted = false;

agentTimer->stop();

emit agentWon();

return;

}

// Получаем следующую позицию из пути

Point nextPos = path[currentPathIndex];

// Проверяем, является ли следующая позиция допустимой

if (Game\_agent\_Maze[nextPos.x][nextPos.y] != 1) {

// Проверяем, достигнет ли агент цели

bool reachedGoal = (Game\_agent\_Maze[nextPos.x][nextPos.y] == 3);

// Перемещаем агента

Game\_agent\_Maze[currentY][currentX] = 0; // Очищаем текущую позицию

Game\_agent\_Maze[nextPos.x][nextPos.y] = 2; // Устанавливаем агента на новую позицию

fillScreenImages(); // Обновляем отображение

currentPathIndex++; // Переходим к следующей точке пути

if (reachedGoal) {

agentStarted = false;

agentTimer->stop();

emit agentWon();

}

} else {

// Если путь заблокирован, пересчитываем его

path.clear();

}

}

void startAgent() {

if (!agentStarted) {

agentStarted = true;

// Запускаем таймер

if (!agentTimer->isActive()) {

agentTimer->start(100); // Движение каждые 500 мс

}

}

}

Функции startAgent() и movePlayer(int, int ) ялвляются методами логика автоматического передвижения и начала движения игрового агета. В movePlayer() описан сигнал AgentWon, который определяет действия после победы в qml.

# Описание программы

При запуске программы происходит запуск основного окна программы. Пользователю доступно три функциональные кнопки *Generate*, *Exit*, *Info* (Таблица 1). После генерации лабиринта пользователю также становятся доступны клавиши стрелки для движения игровой модели(Таблица 2). После нажатия клавиши стрелки каждый раз происходит отображение изменной матрицы лабиринта. В момент нажатия первой клавиши со стрелкой происходит активация логики прохождения лабиринта игровым агентом.

Таблица 1. Функциональные кнопки.

|  |  |  |
| --- | --- | --- |
| Функциональная кнопка, вызывающие событие | Действие пользователя | Действие программы |
| Generate | Клик по кнопке | Выполнение логики по генерации лабиринта и отображение двух игровых зон |
| Exit | Клик по кнопке | Завершение программы |
| Info | Клик по кнопке | Отображение всплывающего окна с информацией о использовании программы |

Таблица 2. Клавиша движения.

|  |  |  |
| --- | --- | --- |
| Клавиша со стрелкой | Действие пользователя | Действие программы |
| Клавиша вниз | Нажатие на кнопку | Обработка события при нажатии на кнопку. Изменение матрицы расположения игровой модели |
| Клавиша вверх | Нажатие на кнопку | Обработка события при нажатии на кнопку. Изменение матрицы расположения игровой модели |
| Клавиша влево | Нажатие на кнопку | Обработка события при нажатии на кнопку. Изменение матрицы расположения игровой модели |
| Клавиша вправо | Нажатие на кнопку | Обработка события при нажатии на кнопку. Изменение матрицы расположения игровой модели |

# Заключение

В результате выполнения Курсовой работы, я обобщил знания о языке си++ в QT фраймворке, в честности типы, присущие QT, а также многомодульная связь, а также QML. Мной было выполнено создание игры лабиринт, проходимого пользователем и игровым агентом одновременно, что обобщило полученные навыки создания графических многомодульных приложений, а также ООП в комплексе.

Подробно были изучены алгоритмы заполнения и прохождения лабиринта, а также методы хранения данных, обработки событий и пр., что способствовало выявлению качественного решения.

# Список Литературы

1. Лабиринт(жанр) википедия - <https://ru.wikipedia.org/wiki/Лабиринт_(жанр)>
2. Обработка сигналов из C++ в коде QML - <https://metanit.com/cpp/qt/6.5.php>
3. Qt 6.2 официальный сайт фраймворка - <https://doc.qt.io/qt-6.2>
4. Лабиринт: классификация, генерирование, поиск решений (статья на habr.com)-<https://habr.com/ru/articles/445378/>
5. Информация о откладке в Qt: <https://doc.qt.io/qtcreator/creator-debugging.html>

# Приложение А.Листинг программы

main.qml:

import QtQuick

import content

import QtQuick.Layouts

import QtQuick.Controls 2.12

Window {

Connections {

target: mazeData

function onGameWon() {

my\_Win\_Loader.active = true

mazeData.cleanup();

}

function onAgentWon() {

console.log("Агент достиг цели!")

game\_Agent\_win.active = true

mazeData.cleanup();

}

}

Loader {

id: my\_Win\_Loader

source: "qml\_others/Me\_win.qml"

active: false

}

Loader {

id: myWindowLoader

source: "qml\_others/Information.qml"

active: false

}

Loader {

id: game\_Agent\_win

source: "qml\_others/Game\_agent\_win.qml"

active: false

}

visibility: "FullScreen"

visible: true

title : qsTr("Лабиринт")

color : "#FFDD73"

Rectangle {

id: main\_Gaym\_Screen

width: parent.width

height: parent.height \* 0.8

focus: false

Keys.onPressed: {

switch (event.key) {

case Qt.Key\_Up:

//movePlayer(0, -1); // Движение вверх

movePlayer(-1, 0);

break;

case Qt.Key\_Down:

//movePlayer(0, 1); // Движение вниз

movePlayer(1, 0);

break;

case Qt.Key\_Left:

//movePlayer(-1, 0); // Движение влево

movePlayer(0, -1); // Движение вверх

break;

case Qt.Key\_Right:

//movePlayer(1, 0); // Движение вправо

movePlayer(0, 1); // Движение вниз

break;

}

}

function movePlayer(dx, dy) {

mazeData.movePlayer(dx, dy);

}

Rectangle {

id: verticalLine

width: 2 // Ширина линии

height: parent.height // Высота линии равна высоте родительского элемента

color: "#FFC000" // Цвет линии

anchors {

horizontalCenter: parent.horizontalCenter // Центрируем по горизонтали

top: parent.top // Привязываем к верхнему краю родителя

bottom: my\_Game\_Screen.bottom // Привязываем к нижнему краю родителя

}

z: 1 // Устанавливаем z-индекс, чтобы линия была поверх других элементов

}

Rectangle{

id: my\_Game\_Screen

radius: 1

width: parent.width/2

height: parent.width/2

anchors {

left: parent.left

top: parent.top

}

color: "steelblue"

GridView {

id: myGameScreenGridView

visible: false // Скрываем GridView по умолчанию

anchors.fill: parent

cellWidth: width / mazeData.myGameScreenImages[0].length

cellHeight: height / mazeData.myGameScreenImages.length

model: mazeData.myGameScreenImages // Изменено на все элементы

delegate: Column {

Repeater {

model: modelData

delegate: Image {

source: modelData // Путь к изображению

width: my\_Game\_Screen.width / mazeData.myGameScreenImages[0].length

height: my\_Game\_Screen.height / mazeData.myGameScreenImages.length

fillMode: Image.Stretch

}

}

}

}

Text {

anchors {

top: parent.bottom

horizontalCenter: parent.horisontlalCenter

}

text: "My Game Screen"

color: "green"

}

}

Rectangle{

id: game\_Agent\_Screen

width: parent.width/2

height: parent.width/2

color: "blue"

anchors {

left: my\_Game\_Screen.right // левая граница по правому краю greenRect

//leftMargin: 10

top: parent.top

//topMargin: 20

}

GridView {

id: game\_Agent\_Screen\_Grid\_View

visible: false

anchors.fill: parent

cellWidth: width / mazeData.agentScreenImages[0].length

cellHeight: height / mazeData.agentScreenImages.length

model: mazeData.agentScreenImages

delegate: Column {

Repeater {

model: modelData

delegate: Image {

source: modelData

width: game\_Agent\_Screen.width / mazeData.agentScreenImages[0].length

height: game\_Agent\_Screen.height / mazeData.agentScreenImages.length

fillMode: Image.Stretch

}

}

}

}

Text {

anchors {

top: parent.bottom

horizontalCenter: parent.horisontlalCenter

}

text: "Game Agent Screen"

color: "red"

}

}

}

RowLayout {

anchors {

horizontalCenter: parent.horizontalCenter

bottom: parent.bottom

bottomMargin: 20

}

spacing: 10 // Optional spacing between buttons

Rectangle {

id: generateButton

color: "lightgreen" // Example color

width: 100

height: 50

radius: 5

Text {

text: "Generate"

anchors.centerIn: parent

color: "black"

}

MouseArea {

anchors.fill: parent

onClicked: {

mazeData.logic\_generate\_maze();

//mazeData.fillScreenImages();

myGameScreenGridView.visible = true

game\_Agent\_Screen\_Grid\_View.visible = true

main\_Gaym\_Screen.focus = true

}

}

}

Rectangle {

id: exitButton

color: "red"

width: 100

height: 50

radius: 5

Text {

text: "Exit"

anchors.centerIn: parent

color: "white"

}

MouseArea {

anchors.fill: parent

onClicked: {

Qt.quit() // This will close the application. Adjust as necessary.

}

}

}

Rectangle {

id: infoButton

color: "lightblue"

width: 100

height: 50

radius: 5

Text {

text: "Info"

anchors.centerIn: parent

color: "black"

}

MouseArea {

anchors.fill: parent

onClicked: {

myWindowLoader.active = true

}

}

}

}

}

Game\_agent\_win.qml:

import QtQuick

Window {

visible: true

id: me\_win

height: 400

width: 400

Text {

anchors.centerIn: parent

text: "ВЫ ПРОИГРАВШИЙ"

font.pixelSize: 30

color: "red"

}

}

Information.qml:

import QtQuick

Window {

id: myWindow

width: 1000

height: 300

title: "Information"

visible: true

Column {

Text {

text: "На экране вы видите две области: <font font color='steelblue'>левая</font> для вашего прохождения, <font font color='blue'>правая</font> для прохождения Игровым агентом.<br>"

textFormat: Text.RichText

}

Text {

text: "Предназначение игрового интерфейса: кнопка <font font color='green'>Generate </font> предназначена для генерации лабиринта. Кнопка <font font color='red'>Exit</font>- для завершения программы.<br>"

textFormat: Text.RichText

}

Text {

text: "После того, как вы создали лабиринт, конечная точка будет помечена <font font color='green'>зеленым</font>. <br> Используйте клавиши-стрелки для начала прохождения, в этот момент и игровой агент параллельно с вами начнет проходить лабиринт.<br>"

textFormat: Text.RichText

}

Text {

text: "Как только первый из вас достигнет <font font color='green'> точки конца</font>, высветится окошко с победителем"

textFormat: Text.RichText

}

}

}

Me\_win.qml:

import QtQuick

Window {

visible: true

id: me\_win

height: 400

width: 400

Text {

anchors.centerIn: parent

text: "ВЫ ПОБЕДИТЕЛЬ"

font.pixelSize: 30

color: "#34D800"

}

}

CmakeList.txt:

cmake\_minimum\_required(VERSION 3.21.1)

#

option(LINK\_INSIGHT "Link Qt Insight Tracker library" ON)

option(BUILD\_QDS\_COMPONENTS "Build design studio components" ON)

project(labirintApp LANGUAGES CXX)

set(CMAKE\_AUTOMOC ON)

find\_package(Qt6 6.2 REQUIRED COMPONENTS Core Gui Qml Quick)

if (Qt6\_VERSION VERSION\_GREATER\_EQUAL 6.3)

qt\_standard\_project\_setup()

endif()

#NEW LINE OF THE ADD IMAGES

set(CMAKE\_AUTORCC ON)

qt\_add\_executable(labirintApp src/main.cpp

src/Logic\_of\_the\_maze.h src/Logic\_of\_the\_maze.cpp

res.qrc

)

qt\_add\_resources(labirintApp "configuration"

PREFIX "/"

FILES

images/game\_agent\_element.jpg

images/my\_game\_element.jpg

images/void\_img.jpg

images/well\_element.jpg

qtquickcontrols2.conf

)

target\_link\_libraries(labirintApp PRIVATE

Qt6::Core

Qt6::Gui

Qt6::Qml

Qt6::Quick

)

set(QT\_QML\_OUTPUT\_DIRECTORY ${CMAKE\_BINARY\_DIR}/qml)

set(QML\_IMPORT\_PATH ${QT\_QML\_OUTPUT\_DIRECTORY}

CACHE STRING "Import paths for Qt Creator's code model"

FORCE

)

#

set(QML2\_IMPORT\_PATH "${CMAKE\_CURRENT\_SOURCE\_DIR}/qml" CACHE STRING "Qt Creator QML Import Path")

if (BUILD\_QDS\_COMPONENTS)

include(${CMAKE\_CURRENT\_SOURCE\_DIR}/qmlcomponents)

endif()

include(${CMAKE\_CURRENT\_SOURCE\_DIR}/qmlmodules)

if (LINK\_INSIGHT)

include(${CMAKE\_CURRENT\_SOURCE\_DIR}/insight)

endif ()

include(GNUInstallDirs)

install(TARGETS labirintApp

BUNDLE DESTINATION .

LIBRARY DESTINATION ${CMAKE\_INSTALL\_LIBDIR}

RUNTIME DESTINATION ${CMAKE\_INSTALL\_BINDIR}

)

# make IDEs aware of the QML import path

set(QML\_IMPORT\_PATH ${PROJECT\_BINARY\_DIR}/qml CACHE PATH

"Path to the custom QML components defined by the project")

res.qml:

<RCC>

<qresource prefix="/qt/qml/Main">

<file>images/game\_agent\_element.jpg</file>

<file>images/my\_game\_element.jpg</file>

<file>images/void\_img.jpg</file>

<file>images/well\_element.jpg</file>

<file>images/end\_element.jpg</file>

<file>qml\_others/Information.qml</file>

<file>qml\_others/Game\_agent\_win.qml</file>

<file>qml\_others/Me\_win.qml</file>

<file>qml\_others/Information.qml</file>

</qresource>

</RCC>