Федеральное государственное автономное образовательное учреждение высшего образования
«Российский университет дружбы народов имени Патриса Лумумбы»
Инженерная академия
Кафедра механики и процессов управления

Курсовая работа

По информатике и программированию

Направление: Прикладная математика и информатика

Профиль: Математические методы механики космического полёта и анализ геоинформационных данных

Тема: Игровые алгоритмы: генерация лабиринтов и поиск пути для NPC **на C++**

Выполняли: Жуков Марк Владимирович, Кирюхин Антон Павлович
Группа: ИПМбд-02-23
№ студенческих: 1132233508, 1132233512

Москва, 2025

Теоретическая справка:

В общем и целом, наша программа демонстрирует ключевые концепции алгоритмов генерации лабиринтов и поиска пути, а также их практическое применение в игровых и исследовательских задачах.

1. Генерация лабиринта

Лабиринт генерируется с использованием алгоритма **Depth-First Search (DFS) с возвратом**. Этот метод гарантирует создание лабиринта с единственным путем между любыми двумя точками, что делает его идеальным для демонстрации алгоритмов поиска пути. Основные шаги алгоритма:

Инициализация: Лабиринт заполняется стенами (#), а все клетки помечаются как непосещенные.
Старт: Алгоритм начинает с заданной стартовой позиции (например, (1, 1)), помечает ее как посещенную и делает проходом ( ).
Рекурсивное построение:

- Из текущей клетки алгоритм случайным образом выбирает направление (вверх, вниз, влево, вправо) и перемещается на две клетки, чтобы избежать "петель".
- Если новая клетка находится в пределах лабиринта и не посещена, стена между текущей и новой клеткой удаляется, а новая клетка добавляется в стек для дальнейшего исследования.
- Если все соседние клетки посещены, алгоритм возвращается к предыдущей клетке (возврат).

Завершение: Процесс продолжается, пока стек не опустеет, что означает полное исследование всех доступных клеток.

2. Поиск пути с использованием алгоритма A*

Алгоритм **A*** — это информированный алгоритм поиска пути, который эффективно находит кратчайший путь между двумя точками, используя эвристическую функцию. Основные компоненты:

Узлы: Каждая клетка лабиринта рассматривается как узел с координатами (x, y), стоимостью пути от старта (g), эвристической оценкой до цели (h) и суммарной стоимостью (f = g + h).
Эвристика: В данной реализации используется **манхэттенское расстояние** — сумма абсолютных разностей координат текущей клетки и цели. Это допустимая эвристика, так как она никогда не переоценивает стоимость пути.
Открытый и закрытый списки:

- Открытый список (приоритетная очередь) содержит узлы, которые нужно исследовать, упорядоченные по возрастанию f.
- Закрытый список (неявный) состоит из уже исследованных узлов.

Процесс поиска:

1. Стартовый узел добавляется в открытый список.
2. На каждом шаге извлекается узел с наименьшей суммарной стоимостью (f).

3. Если узел является целевым, путь восстанавливается по ссылкам на родительские узлы.
4. Соседние клетки (в 4 направлениях) проверяются на проходимость. Для каждой допустимой соседней клетки вычисляется новая стоимость g. Если клетка еще не исследована или найден более короткий путь, она добавляется в открытый список.

Визуализация: Во время поиска текущее состояние лабиринта отображается в терминале с задержкой, чтобы показать процесс исследования клеток и построения пути.

3. Взаимодействие классов

Класс Maze:

- Хранит структуру лабиринта в виде матрицы символов.
- Предоставляет методы для генерации лабиринта, проверки проходимости клеток и отметки пути.

Класс AStarPathfinder:

- Использует экземпляр Maze для доступа к данным лабиринта.
- Реализует алгоритм A* для поиска пути, используя методы Maze для проверки клеток и визуализации пути.

Связь между классами: AStarPathfinder объявлен дружественным классом к Maze, что позволяет ему напрямую обращаться к приватным данным лабиринта, таким как матрица grid.

Решение (с комментариями):

maze > C Maze.h

```h
#ifndef MAZE_H
#define MAZE_H

#include <vector>
#include <iostream>

// Размеры лабиринта по умолчанию
constexpr int WIDTH = 21;
constexpr int HEIGHT = 21;

// Класс Maze отвечает за генерацию и хранение структуры лабиринта
class Maze {
private:
    // grid — матрица символов ('#' — стена, ' ' — проход, '0' — путь)
    std::vector<std::vector<char>> grid;

    // visited — матрица флагов посещения для генерации
    std::vector<std::vector<bool>> visited;

    // Направления движения: вниз, вверх, вправо, влево (двигаемся на 2 клетки)
    const int dx[4] = {0, 0, 2, -2};
    const int dy[4] = {2, -2, 0, 0};

    // Проверка, находится ли точка внутри границ лабиринта
    bool isInBounds(int x, int y) const;

public:
    Maze();
    void generate(int startX, int startY); // Генерация лабиринта с указанной стартовой точки
    bool isWalkable(int x, int y) const;   // Проверка, можно ли пройти по клетке
    void markPath(int x, int y);           // Отметить клетку как часть пути
    void print() const;                    // Напечатать лабиринт

    friend class AStarPathfinder; // Доступ к приватным данным для поиска пути
};

#endif
```

maze > Maze.cpp

```cpp
#include "Maze.h"
#include <stack>
#include <algorithm>
#include <random>

// Конструктор: инициализирует сетку лабиринта символами '#' и массив visited значениями false
Maze::Maze() {
    grid = std::vector<std::vector<char>>(HEIGHT, std::vector<char>(WIDTH, '#'));
    visited = std::vector<std::vector<bool>>(HEIGHT, std::vector<bool>(WIDTH, false));
}

// Проверка, что координаты (x, y) находятся в пределах границ лабиринта
bool Maze::isInBounds(int x, int y) const {
    return x >= 0 && x < HEIGHT && y >= 0 && y < WIDTH;
}

// Генерация лабиринта методом DFS с возвратом
void Maze::generate(int startX, int startY) {
    std::stack<std::pair<int, int>> stack; // stack это структура данных реализующая FILO
    stack.push({startX, startY});
    visited[startX][startY] = true;
    grid[startX][startY] = ' ';

    std::random_device rd;
    std::mt19937 g(rd()); // генератор случайных чисел

    while (!stack.empty()) {
        auto [x, y] = stack.top();
        std::vector<int> dirs = {0, 1, 2, 3};
        std::shuffle(dirs.begin(), dirs.end(), g); // случайный порядок направлений

        bool moved = false;
        for (int i : dirs) {
            int nx = x + dx[i]; // новое направление
            int ny = y + dy[i];

            if (isInBounds(nx, ny) && !visited[nx][ny]) {
                visited[nx][ny] = true;
                grid[nx][ny] = ' ';

                grid[x + dx[i] / 2][y + dy[i] / 2] = ' ';

                // продолжаем путь
                stack.push({nx, ny});
                moved = true;
                break;
            }
        }

        // если некуда идти — откатываемся назад
        if (!moved) stack.pop();
    }
}

// Проверка, можно ли пройти по заданной клетке (т.е. она не стена)
bool Maze::isWalkable(int x, int y) const {
    return isInBounds(x, y) && grid[x][y] == ' ';
}

// Отметить путь на карте (для анимации и отрисовки)
void Maze::markPath(int x, int y) {
    grid[x][y] = '0'; // путь обозначается нулём
}

// Отображение лабиринта в терминал
void Maze::print() const {
    for (const auto& row : grid) {
        for (char c : row)
            std::cout << c;
        std::cout << '\n';
    }
}
```

maze > C AStarPathfinder.h

```h
#ifndef ASTARPATHFINDER_H
#define ASTARPATHFINDER_H

#include "Maze.h"

// Структура узла
struct Node {
    int x, y;      // координаты
    float g, h;    // g — путь от старта, h — эвристика до цели
    Node* parent;  // ссылка на родителя (для восстановления пути)

    float f() const { return g + h; }
};

// Класс AStarPathfinder реализует алгоритм A* для поиска пути по лабиринту
class AStarPathfinder {
private:
    Maze& maze;

    // Направления движения: вверх, вниз, влево, вправо
    const int dx[4] = {0, 0, 1, -1};
    const int dy[4] = {1, -1, 0, 0};

    // Эвристическая функция — манхэттенское расстояние
    float heuristic(int x1, int y1, int x2, int y2);

public:
    AStarPathfinder(Maze& m);
    void findPath(int startX, int startY, int goalX, int goalY); // поиск пути и анимация
};

#endif
```

maze > AStarPathfinder.cpp

```cpp
#include "AStarPathfinder.h"
#include <queue>
#include <map>
#include <cmath>
#include <thread>
#include <chrono>
#include <iostream>

// Конструктор принимает ссылку на лабиринт
AStarPathfinder::AStarPathfinder(Maze& m) : maze(m) {}

// Манхэттенское расстояние как эвристика
float AStarPathfinder::heuristic(int x1, int y1, int x2, int y2) {
    return std::abs(x1 - x2) + std::abs(y1 - y2);
}

// Алгоритм A*: находит путь и визуализирует его
void AStarPathfinder::findPath(int startX, int startY, int goalX, int goalY) {
    // Компаратор для приоритетной очереди — выбирает узел с наименьшим f()
    auto cmp = [](Node* a, Node* b) { return a->f() > b->f(); };
    std::priority_queue<Node*, std::vector<Node*>, decltype(cmp)> openList(cmp);

    // Храним указатели на все созданные узлы (чтобы потом удалить)
    std::map<std::pair<int, int>, Node*> allNodes;

    // Стартовый узел
    Node* start = new Node{startX, startY, 0, heuristic(startX, startY, goalX, goalY), nullptr};
    openList.push(start);
    allNodes[{startX, startY}] = start;

    // Основной цикл A*
    while (!openList.empty()) {
        Node* current = openList.top();
        openList.pop();

        // Если достигли цели — строим путь и анимируем
        if (current->x == goalX && current->y == goalY) {
            while (current) {
                maze.markPath(current->x, current->y); // отмечаем путь
```

```
                // Очищаем экран и рисуем лабиринт
                std::cout << "\x1B[2J\x1B[H";
                maze.print();

                std::this_thread::sleep_for(std::chrono::milliseconds(100)); // задержка
                current = current->parent; // идем к предыдущему узлу
            }
            break;
        }

        // Проверка соседей (в 4 направлениях)
        for (int i = 0; i < 4; ++i) {
            int nx = current->x + dx[i];
            int ny = current->y + dy[i];

            if (!maze.isWalkable(nx, ny)) continue;

            float newG = current->g + 1;
            auto key = std::make_pair(nx, ny);

            // Если узел ещё не был создан или найден более короткий путь
            if (allNodes.find(key) == allNodes.end() || newG < allNodes[key]->g) {
                Node* neighbor = new Node{nx, ny, newG, heuristic(nx, ny, goalX, goalY), current};
                allNodes[key] = neighbor;
                openList.push(neighbor);
            }
        }
    }

    // Очистка всех узлов из памяти
    for (auto& pair : allNodes)
        delete pair.second;
}
```

maze > main.cpp

```
#include <cstdlib>
#include <ctime>
#include "Maze.h"
#include "AStarPathfinder.h"

int main() {
    // Инициализация генератора случайных чисел текущим временем
    // Это нужно для получения разных лабиринтов при каждом запуске
    srand(static_cast<unsigned>(time(0)));

    // Создание лабиринта и его генерация от стартовой позиции (1, 1)
    Maze maze;
    maze.generate(1, 1);

    // Создание объекта поиска пути и запуск поиска от (1,1) до (HEIGHT-2, WIDTH-2)
    // Это почти правый нижний угол, но на 1 клетку отступаем, чтобы не упереться в границу
    AStarPathfinder pathfinder(maze);
    pathfinder.findPath(1, 1, HEIGHT - 2, WIDTH - 2);

    // Повторный вывод лабиринта после завершения анимации
    // (финальное состояние с нарисованным маршрутом)
    maze.print();
    return 0;
}
```