Комп’ютерний практикум № 04

Контейнери з послідовним доступом. Ітератори. Сортування. Бінарний пошук.

Рівень “Високий”

Хід роботи

**Завдання 1**

*Розробити модель ігрового поля у вигляді квадратної матриці 20х20 за наступними правилами:*

* *модель повинна бути реалізована в окремому модулі за допомогою класу і підтримувати читання та запис в бінарний файл.*
* *при розробці моделі враховувати, що в майбутньому кожна клітинка може містити певний ігровий об’єкт (наприклад, вежу, чи перешкоду).*
* *три точки входу ворогів, відповідно за час роботи програми повинні з’явитись три вороги по одному з кожної точки входу.*

class ModelMap {

public:

UINT width;//width

UINT height;//height

cell \*\*area; //cell type

COORD castle;

//COORD entry;

vector<enemy> enemies;

vector<tower> towers;

void setCell(int i, int j, cell c){ area[i][j] = c;}

cell getCell(int i, int j ){return area[i][j]; }

ModelMap(const int w, const int h);

ModelMap(string fileName);

ModelMap(const ModelMap & ModMap);

~ModelMap();

void drawMap(sf::RenderWindow &window);

void drawPath(sf::RenderWindow &window, list<COORD> shortWay);

void StartGame(sf::RenderWindow &window, unsigned int \*tickCount);

void defineEfficiency();

};

**Завдання 2**

* *додати в модель карти шлях руху ворога від точки входу (точка входу - одна) до бази гравця за допомогою std::vector.*

*- при запуску програми один ворог повинен з’являтись в точці входу і проходити по заданому маршруту до бази гравця.*

*- шлях ворога повинен будуватись автоматично з застосуванням алгоритму пошуку найкоротшого шляху (можна реалізувати будь-який алгоритм);*

*- на клітинках, пройдених ворогом повинні малюватись сліди; - шлях повинен зберігатись в std::list; - для кожної точки входу повинен вираховуватись свій шлях.*

|  |  |
| --- | --- |
| **D:\Studing\Пары\Програмування -2. Алгоритми і структури даних\Звіт\Lialgoritm.png**  **D:\Studing\Пары\Програмування -2. Алгоритми і структури даних\Звіт\Lialgoritm.png** | FindPath::FindPath(int sizeX, int sizeY, COORD endP, vector<vector<cell>> mapP) : width(sizeX), height(sizeY){    end.X = endP.X;  end.Y = endP.Y;  map = mapP;  visited.resize(height);  path.resize(height);  for(int i=0; i<height; i++){  visited[i].resize(width);  path[i].resize(width);  }  clearMap();  }  void FindPath::findShortWay(enemy& unit){    clearMap();  begin.X = unit.entry.X;  begin.Y = unit.entry.Y;  plan.push(begin);  path[begin.X][begin.Y] = 1;  while(!plan.empty()){  COORD cur = plan.front();  plan.pop();  find\_path( cur.X, cur.Y);    }  recoverPath();  shortWay.reverse();  swap(shortWay, unit.shortWay);  unit.setposition(unit.shortWay.begin());  }  void FindPath::find\_path( short row, short col){  if(!visited[row][col]){  COORD cur ;    if ((row+1) < height && (row+1) >= 0 && !visited[row+1][col] &&  (map[row+1][col].type == ROAD || map[row+1][col].type == BASE)) {  path[row+1][col] = path[row][col] + 1;  cur.X=row+1;  cur.Y=col;  plan.push(cur);  }  if((row-1) < height && (row-1) >= 0 && !visited[row-1][col] &&  (map[row-1][col].type == ROAD || map[row-1][col].type == BASE)) {  path[row-1][col] = path[row][col] + 1;  cur.X=row-1;  cur.Y=col;  plan.push(cur);  }  if((col + 1) < width && (col + 1) >= 0 && !visited[row][col+1] &&  (map[row][col+1].type == ROAD || map[row][col+1].type == BASE)) {  path[row][col+1] = path[row][col] + 1;  cur.X=row;  cur.Y=col+1;  plan.push(cur);  }  if((col - 1) < width && (col - 1) >= 0 && !visited[row][col-1] &&  (map[row][col-1].type == ROAD || map[row][col-1].type == BASE)) {  path[row][col-1] = path[row][col] + 1;  cur.X=row;  cur.Y=col-1;  plan.push(cur);  }  visited[row][col] = 1; /\* отмечаем клетку в которой побывали \*/  //cout<<" path "<<path[row][col]<<endl;  }}  void FindPath::recoverPath() {  if(!visited[end.X][end.Y]){  cout << "Not way" << endl;  }  else {  short x = end.X;  short y = end.Y;  shortWay.push\_back(end);  while (path[x][y] != 2) { // восстановление пути\*/  if ((x-1) >= 0 && (x-1) < width && (path[x-1][y] == path[x][y] - 1)) {  shortWay.push\_back(COORD{--x,y});  }  else if ((x+1) >= 0 && (x+1) < width && (path[x+1][y] == path[x][y] - 1)) {  shortWay.push\_back(COORD{++x,y});  }  else if ((y-1) >= 0 && (y-1) < height && (path[x][y-1] == path[x][y] - 1)) {  shortWay.push\_back(COORD{x,--y});  }  else if ((y+1) >= 0 && (y+1) < height && (path[x][y+1] == path[x][y] - 1)) {  shortWay.push\_back(COORD{x,++y});  }}}}  void FindPath::clearMap(){  for(int i=0; i<width; i++){  for(int j=0; j<height; j++){  visited[i][j] = 0;  path[i][j] = -1;  }}} |

**Завдання 3**

* *Відсортувати клітинки, на якій можна побудувати захисну вежу по спаданню “ефективності”.*

|  |  |
| --- | --- |
|  | void ModelMap::defineEfficiency(){  int x, y, rad;  for(int i= 0; i < towers.size();i++){  x = towers[i].coord.X;  y = towers[i].coord.Y;  rad = 2;  for(int Xcoord = -rad; Xcoord <= rad; Xcoord++){  for(int Ycoord = -rad; Ycoord <= rad; Ycoord++){  if((x+Xcoord) < width && (y+Ycoord) < height && (x+Xcoord) > 0 && (y+Ycoord) > 0 && field[x+Xcoord][y+Ycoord].type == ROAD){  for(int unit = 0; unit < enemies.size(); unit++){  for(list<COORD>::iterator cell = enemies[unit].short\_way.begin(); cell != enemies[unit].short\_way.end(); cell++){  if((\*cell).X == (x+Xcoord) && (\*cell).Y == (y+Ycoord)) towers[i].efficiency++;  }  }  }  }  }  } |

|  |
| --- |
| **Makefile**  all: solve  solve: main.o CreateMap.o FindPath.o          g++ main.o CreateMap.o FindPath.o -o main -L "C:Program Files (x86)/SFML-2.4.0/lib" -lsfml-graphics-d -lsfml-window-d -lsfml-system-d  main.o: main.cpp          g++ -c main.cpp -I"C:/Program Files (x86)/SFML-2.4.0/include"  CreateMap.o: CreateMap.cpp          g++ -c CreateMap.cpp -I "C:/Program Files (x86)/SFML-2.4.0/include"  FindPath.o: FindPath.cpp          g++ -c FindPath.cpp -I"C:/Program Files (x86)/SFML-2.4.0/include"  clean:      del -rf \*.o \*.exe |

**Висновок:** в рамках роботи над комп’ютерним практикумом я навчилася використовувати контейнери з послідовним доступом, ітератори, реалізувала алгоритм хвильового пошуку для пошуку найкоротшого шляху ворога.