기말 프로젝트: Maze generator visualization

20160768 김홍엽

1. 프로젝트 목표: Maze generator visualization

DFS 알고리즘을 활용한 미로 생성과정을 실시간으로 시각화하여 사용자에게 보여준다.

2. 프로젝트 설명

이 프로젝트는 기존 미로 생성 알고리즘인 Eller's Algorithm을 사용하지 않고, DFS 알고리즘을 사용하여 미로를 생성하였다. 또한 maze generator(빨간 원)가 움직이는 과정을 실시간으로 보여줌으로써 미로 생성 과정을 시각화하였다. 사용자가 미로의 크기를 자유롭게 변경할 수 있도록 하여, 다양한 크기의 미로 생성 과정을 시각적으로 체험할 수 있다. 이를 통해 사용자는 DFS 알고리즘이 미로 생성시에 어떤 식으로 작동하는 지 쉽게 이해할 수 있다.

3. 자료구조 및 변수 설명

3-1. class Cell

```
class Cell { // 미로의 셀을 구성하는 클래스
private:
    int row; // 셀의 row 값(y좌표)
    int column; // 셀의 column 값(x좌표)
    bool walls[4]; // 셀의 상하좌우 벽
    bool visited; // 셀 방문 여부
    int direction[4] = {0, 1, 2, 3}; // 0: top, 1: right, 2: bottom, 3: left

public:
    Cell(int row, int column){
        this->row = row;
        this->column = column;
        visited = false;

        for (int i = 0; i < 4; i++){ // 셀의 상하좌우 벽을 세운다.
            walls[i] = true;
        }
    };

    void setVisited(bool b){ // 방문 표시
        visited = b;
    }

    bool isVisited(){ // 방문 여부 확인
        return visited;
    }

    int getRow(){ // 셀의 row 값 return
        return row;
    }

    int getColumn(){ // 셀의 column 값 return
        return column;
    }

    bool *getWalls(){ // 셀의 벽 정보 return
        return walls;
    }

      void connectCell(Cell &next){ // 파라미터로 받은 셀과 연결한다.
          if (column > next.column) { // 다음 셀이 현재 셀의 왼쪽에 위치
            walls[3] = false; // 현재 셀의 왼쪽 벽을 제거한다.
            next.walls[1] = false; // 다음 셀의 오른쪽 벽을 제거한다.
          } else if (column < next.column) { // 다음 셀이 현재 셀의 오른쪽에 위치
            walls[1] = false; // 현재 셀의 오른쪽 벽을 제거한다.
            next.walls[3] = false; // 다음 셀의 왼쪽 벽을 제거한다.
          } else if (row > next.row) { // 다음 셀이 현재 셀의 위쪽에 위치
            walls[0] = false; // 현재 셀의 위쪽 벽을 제거한다.
            next.walls[2] = false; // 다음 셀의 아래쪽 벽을 제거한다.
          } else if (row < next.row) { // 다음 셀이 현재 셀의 아래쪽에 위치
            walls[2] = false; // 현재 셀의 아래쪽 벽을 제거한다.
            next.walls[0] = false; // 다음 셀의 위쪽 벽을 제거한다.
          }
      }
  };
```

Cell은 미로의 셀을 구성하는 클래스이다. 각 셀은 row(y좌표값), column(x좌표값), walls(셀의 상하좌우 벽), visited(셀 방문 여부), direction(상하좌우)의 변수를 갖는다.
Cell 클래스의 함수로는 생성자, setVisited, isVisited, getRow, getColumn, getWalls, connectCell이 있다.

함수	설명
생성자	row와 column, visited를 초기화하고, 상하좌우의 벽을 세운다.
setVisited	셀의 방문 여부를 표시한다.
isVisited	해당 셀의 방문 여부를 반환한다.
getRow	셀의 row 값을 반환한다.
getColumn	셀의 column 값을 반환하다.
getWalls	셀의 walls를 반환하다.
connectCell	해당 셀과 매개변수로 받은 셀을 연결한다. 연결이라는 의미는 셀과 셀을 가로막고 있는 벽을 제거하는 것이다.

3-2. 변수

```
float windowWidth, windowHeight;

bool start_flag; // 시작 flag
bool draw_flag; // draw flag

int width; // 미로의 넓이
int height; // 미로의 높이
int cell_size; // 셀의 크기

vector<Cell> maze; // Cell로 이루어진 미로 정보
stack<Cell *> s; // DFS를 위한 스택
Cell *current; // 현재 셀
```

변수	설명
float windowWidth, windowHeight	윈도우창의 넓이와 높이를 저장하는 변수이다.
bool start_flag	프로그램의 시작 flag 변수이다.
bool draw_flag	프로그램의 그리기 flag 변수이다.
int width	미로의 넓이를 저장하는 변수이다.
int height	미로의 높이를 저장하는 변수이다.
int cell_size	셀 하나의 크기를 저장하는 변수이다. 이 변수를 통해 그려지는 미로의 크기를 조절할 수 있다.
vector<Cell> maze	미로 정보를 저장하기 위한 변수이다. 클래스 Cell로 이루어진 벡터 자료구조를 사용하였다. 미로의 정보는 2차원이 아닌 1차원으로 저장된다.
Stack<Cell *> s	DFS 알고리즘을 위한 스택이다. 미로 생성시에 사용된다.
Cell *current	현재 셀을 저장하기 위한 변수이다. 미로 생성시 maze generator가 현재 위치한 셀이다.

4. 프로젝트 함수 및 알고리즘 설명: DFS 알고리즘을 활용한 미로 생성기

void ofApp::setup()

```
void ofApp::setup(){
    ofSetFrameRate(15);
    ofBackground(255, 255, 255);

    start_flag = false;
    draw_flag = true;

    height = 20;
    width = 20;
    cell_size = 20;

    ofSetWindowShape(width*cell_size*1.2, height*cell_size*1.2); // 윈도우창 사이즈 조절

    windowWidth = ofGetWidth();
    windowHeight = ofGetHeight();

    // 윈도우창 중앙 위치
    ofSetWindowPosition((ofGetScreenWidth()-windowWidth)/2, (ofGetScreenHeight()-windowHeight)/2);

    for (int row = 0; row < height; row++){ // 미로의 높이, 넓이 정보를 활용하여 maze를 초기화한다.
        for (int column = 0; column < width; column++){
            maze.push_back(Cell(row, column));
        }
    }
    current = &maze[0]; // maze[0](cell(0,0))을 현재 셀로 초기화한다.

}
```

초기 미로의 높이와 넓이, cell_size는 모두 20이다. 미로의 높이, 넓이 정보를 활용하여 maze를 초기화하는 작업을 진행한 후, maze[0](Cell(0,0))을 현재 셀로 초기화한다.

void ofApp::update()

```
void ofApp::update(){
    /*
     미로를 생성하는 함수이다.
     */
    if (start_flag == true){
        int speed = 1; // draw 속도(미로 탐색 속도)를 조절하기 위한 변수, 값이 증가할수록 화면이 보이는 draw 속도가 빨라진다.
        while(speed > 0){
            current->setVisited(true); // 현재 셀을 방문 표시한다
            Cell *next = findNextCell(); // 현재 셀을 기준으로 다음 방문할 셀을 찾는다.

            // 방문할 셀이 존재한다면 방문 표시를 한 뒤,
            // 스택에 현재 셀을 push하고 현재 셀과 다음 셀을 연결한다.
            // 현재 셀을 다음 셀로 이동한다.
            if (next != NULL){
                next->setVisited(true);
                s.push(current);
                current->connectCell(*next);
                current = next;
            }
            else if (s.size() > 0){ // 스택이 비어있지 않다면, 현재 셀을 스택의 top으로 이동한다.
                current = s.top();
                s.pop();

            } else if (s.size() == 0){ // 스택의 사이즈가 0이라면 탐색이 종료된다.
                break;
            }

            speed--;
        }
    }
}
```

미로 생성은 start_flag가 set되었을 때, update() 함수에서 이루어진다. speed 변수는 미로가 그려지는 속도를 조절하기 위해 설정한 변수이다.
먼저 current(현재 위치한 셀)를 방문 표시한다. 그리고 findNextCell() 함수를 통해 current를 기준으로 다음 방문할 셀을 탐색한 후, next 변수에 저장한다.
next가 존재할 경우 next를 방문 표시한 후, current를 스택에 push한다. current와 next를 connectCell() 함수를 통해 연결한 후, current를 next로 이동한다.
next가 존재하지 않고, 스택이 비어있지 않다면, current를 스택의 top으로 이동한다.
next가 존재하지 않고, 스택도 비어있다면 탐색을 종료한다.

Cell *ofApp::findNextCell()

```
Cell *ofApp::findNextCell(){
    /*
    현재 셀(current)을 기준으로 방문 가능한 다음 셀을 찾는 함수이다.
    Cell의 pointer를 반환한다.
     */
    vector<Cell *> neighbors; // 이웃 셀의 정보를 담을 vector를 초기화한다.
    int currentRow = current->getRow();
    int currentColumn = current->getColumn();

    int neighborPos[4]; // 이웃 셀들의 위치 정보
    neighborPos[0] = getPos(currentRow - 1, currentColumn);
    neighborPos[1] = getPos(currentRow, currentColumn - 1);
    neighborPos[2] = getPos(currentRow + 1, currentColumn);
    neighborPos[3] = getPos(currentRow, currentColumn + 1);

    for (int i = 0; i < 4; i++){ // 이웃 셀들 중에서 방문이 가능하고, 아직 방문하지 않은 셀들을 neighbors에 저장한다.
        if(neighborPos[i]!= -1 && !maze[neighborPos[i]].isVisited()){
            neighbors.push_back(&maze[neighborPos[i]]);
        }
    }

    if(neighbors.size() > 0){ // neighbors 중에서 방문할 다음 셀을 랜덤하게 선택한다.
        return neighbors.at(rand() % neighbors.size());
    }

    return NULL; // 방문 가능한 이웃 셀이 없다면 NULL을 반환한다.
}
```

current(현재 셀)을 기준으로 방문 가능한 다음 셀을 찾는 함수이다. 이웃 셀의 정보를 담을 vector를 선언한다. current의 row와 column 값을 얻은 후, 그 값을 기준으로 상하좌우 셀들의 위치 정보를 getPos() 함수를 통해 얻는다. 이웃 셀들 중에서 방문이 가능하고, 아직 방문하지 않은 셀들을 neighbors에 저장한다. neighbors에 셀들이 존재한다면, 그것들 중 다음 이동할 셀을 임의로 선택하여 반환한다. 방문 가능한 이웃 셀이 없다면 NULL을 반환한다.

int ofApp::getPos(int row, int column)

```
int ofApp::getPos(int row, int column){
    /*
     파라미터로 받은 row, column 값이 maze내 존재 가능한지 파악한다.
     존재할 수 없는 위치이면, -1을 반환하고, 존재 가능하다면, 셀의 인덱스를 반환한다.
     */
    if(row < 0 || column < 0 || column > width -1 || row > height - 1)
        return -1;
    else
        return column + row * width; // maze는 일차원으로 저장되어 있기 때문에 인덱스 조정이 필요하다.
}
```

매개변수로 받은 row, column 값이 maze 내 존재 가능한지 파악한다. 존재할 수 없는 위치이면, -1을 반환한다. 존재 가능하다면, column + row * width 값을 반환한다. column + row * width을 반환하는 이유는 maze가 1차원으로 저장되어 있기 때문에 이에 맞추어 셀의 위치를 반환하기 위해서이다. (2차원의 위치를 1차원으로 매핑하는 것으로 이해하면 된다.)

void ofApp::draw()

```
void ofApp::draw(){
    if (draw_flag == true){
        // 윈도우 사이즈에 맞추어 미로의 크기를 조절한다.
        while(height * cell_size >= ofGetHeight() || width * cell_size >= ofGetWidth()){
            cell_size = cell_size * 0.9;
        }
        while(height * cell_size <= ofGetHeight() * 0.8 && width * cell_size <= ofGetWidth() * 0.8){
            cell_size = cell_size * 1.1;
        }

        float x1, y1, x2, y2;
        size_t i;
        int w = 0;
        int h = 0;

        ofSetColor(ofColor::black);
        ofSetLineWidth(cell_size*0.1);

        // 미로 정보를 통해 미로를 그린다. (line 사용)
        for (i = 0; i < maze.size(); i++){
            w = i % width;
            bool *walls = maze[i].getWalls();
            if (walls[0]){ // 위쪽 벽을 그린다.
                x1 = w * cell_size;
                x2 = (w+1) * cell_size;
                y1 = h * cell_size;
                y2 = h * cell_size;
                ofDrawLine(x1, y1, x2, y2);
            }
            if (walls[1]){ // 오른쪽 벽을 그린다.
                x1 = (w+1) * cell_size;
                x2 = x1;
                y1 = h * cell_size;
                y2 = (h+1) * cell_size;
                ofDrawLine(x1, y1, x2, y2);
            }
            if (walls[2]){ // 아래쪽 벽을 그린다.
                x1 = w * cell_size;
                x2 = (w+1) * cell_size;
                y1 = (h+1) * cell_size;
                y2 = (h+1) * cell_size;
                ofDrawLine(x1, y1, x2, y2);
            }
            if (walls[3]){ // 왼쪽 벽을 그린다.
                x1 = w * cell_size;
                x2 = x1;
                y1 = h * cell_size;
                y2 = (h + 1) * cell_size;
                ofDrawLine(x1, y1, x2, y2);
            }

            if((i+1) % width == 0){
                h = h + 1;
            }
        }

        // maze generator 시각화
        // 빨간 동그라미가 움직이며 실시간으로 미로를 생성한다. (DFS의 탐색 경로)
        int current_y = current->getRow();
        int current_x = current->getColumn();
        ofSetColor(ofColor::red);
        ofDrawCircle((current_x*cell_size + (current_x+1)*cell_size)/2, (current_y*cell_size + (current_y+1)*cell_size)/2,
            cell_size*0.2);
    }

}
```

미로와 maze generator(빨간 원)를 그리는 함수이다. 윈도우 사이즈에 따라 cell_size를 조절하여 미로의 크기를 자동적으로 조절한다. ofDrawLine을 통해 각 셀의 벽을 그려줌으로써 미로를 시각적으로 보여준다.

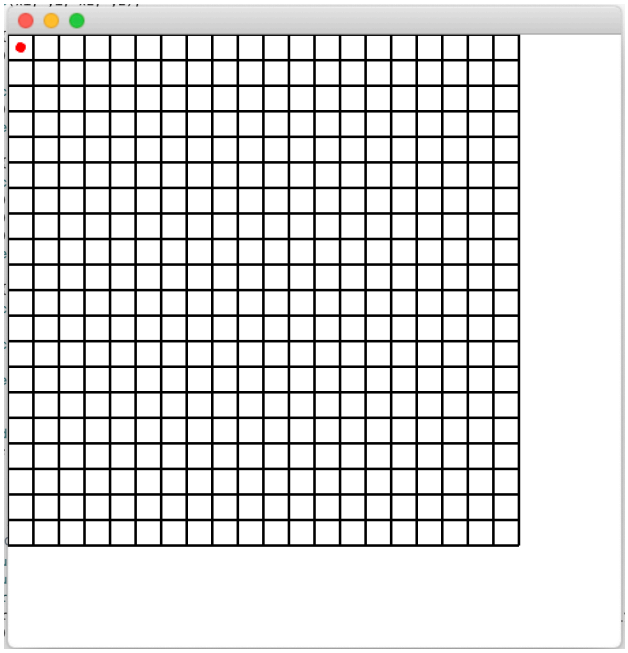
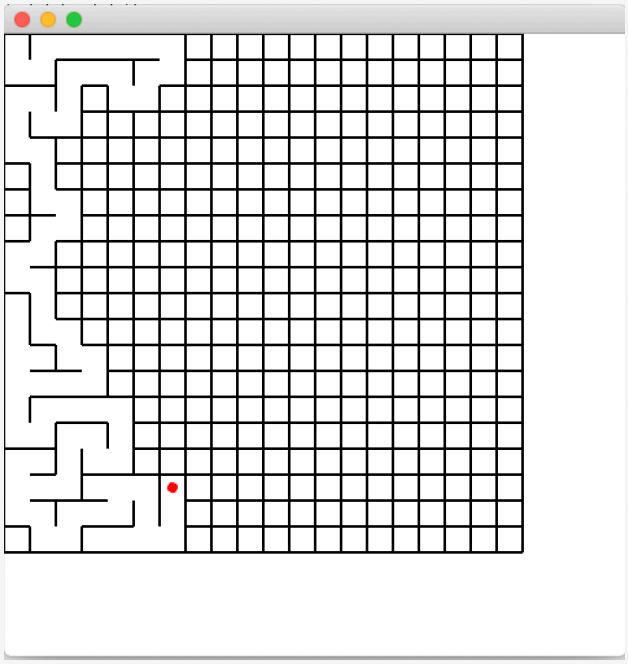
current 변수의 위치 정보를 얻어 현재 current 셀에 위치한 maze generator를 실시간으로 보여준다. 이를 통해 DFS 알고리즘의 탐색 경로를 시각적으로 체험할 수 있다.

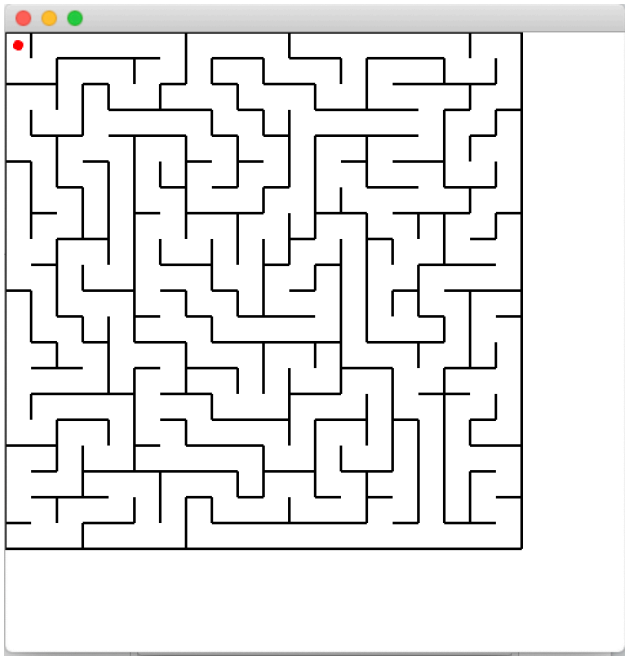
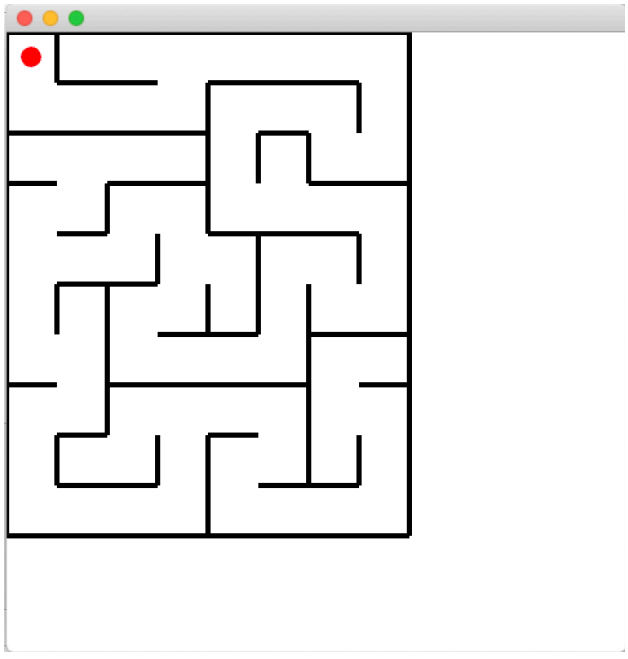
void ofApp::keyPressed(int key)

```
void ofApp::keyPressed(int key){
    if (key == 's'){ // 미로 생성 시작
        start_flag = true;
    }

    if (key == 'p'){ // 미로 생성 일시정지
        start_flag = false;
    }

    if (key == 'q'){ // 프로그램 종료 및 메모리 할당 해제
        draw_flag = false;

        while(!s.empty()) s.pop();
        maze.clear();
        maze.shrink_to_fit();

        cout << "Dynamically allocated memory has been freed." << endl;
        _Exit(0);
    }

    if (key == 'c'){ // 미로 정보 변경 (사용자 입력)
        cout << "Height of Maze: ";
        cin >> height;
        cout << "width of Maze: ";
        cin >> width;

        // 미로의 최대 높이, 넓이 제한
        if (height > 40) height = 40;
        if (width > 40) width = 40;

        start_flag = false;
        draw_flag = false;

        maze.clear();
        maze.shrink_to_fit();
        while(!s.empty()) s.pop();

        for (int row = 0; row < height; row++){
            for (int column = 0; column < width; column++){
                maze.push_back(Cell(row, column));
            }
        }
        current = &maze[0];

        start_flag = true;
        draw_flag = true;

    }
}
```

’s’ 키를 누르면 start_flag가 set 된다.
‘p’ 키를 누르면 start_flag가 false가 된다.
‘q’ 키를 누르면 draw_flag가 false가 되고, 할당된 메모리가 해제된 뒤, 프로그램이 종료된다.
‘c’ 키를 누르면, 사용자가 미로 정보를 변경할 수 있다. 미로의 최대 높이와 넓이는 40으로 제한하였다. 새롭게 업데이트된 크기로 미로가 초기화되고, 미로 생성이 시작된다.

5. 프로그램 실행 화면

프로그램 첫 실행 화면	
's' 키를 눌러 미로 생성을 시작한 화면	

미로 생성을 완료한 화면	
‘c’ 키를 눌렀을 때 나타나는 콘솔 창	2021-06-21 17:55:41.335914+0900 comsilProjectDebug[24631:1257463] Metal API Validation Enabled Height of Maze: 10 width of Maze: 8
10x8의 미로로 변경되어 미로 생성을 완료한 화면	

6. 느낀 점 및 개선 사항

기존 실습에서 미로 경로 탐색 때 사용되었던 DFS 알고리즘을 미로 생성에도 적용할 수 있음을 알게 되었다. 이처럼 한 가지 알고리즘이 여러 다양한 유형의 문제에도 적용될 수 있기에, 여러 주요 알고리즘을 익히는 것이 다양한 문제를 효율적으로 해결하는데 도움이 될 수 있다는 것을 느꼈다.

이 프로젝트의 주요 목표 중 하나는 사용자가 미로 생성 과정을 시각적으로 체험할 수 있도록 돕는 것이다. 현재는 빨간 원이 이동하는 과정을 보여줌으로써 미로 생성 과정을 표현하였는데, 정적인 빨간 원 대신 움직이는 하나의 캐릭터로 표현하였더면 조금 더 유쾌한 표현이 가능했을 것이라는 생각이 들었다. 또한 DFS 뿐만 아니라 다양한 알고리즘을 추가하여 사용자가 미로 생성 알고리즘을 직접 선택할 수 있도록 프로젝트를 개선해보아도 좋을 것 같다.