최종 프로젝트 보고서

20201559 김민준

1. 프로젝트 목표

   실습 시간에 구현한 BFS, DFS를 시각화 하여 BFS, DFS알고리즘에 대한 이해를 돕는다. 또한 기존 실습에서는 이미 등록된 .maz 파일에 대한 탐색만 가능하였지만 이 프로젝트에서는 백트래킹 알고리즘을 통한 새로운 미로를 생성하고 이를 파파일 저장하는 것을 지원한다.

2. 실험 환경

   Visual studio 를 기반으로한 OpenFramework, 4K(3840*2160) 환경에서 실습한 결과이기에 다른 해상도에서 실행시 Openframeworks의 특성상 실험 환경보다 더 크게 화면일 출력될 수 있음, ofApp.h 의 LINEH, LINEW 값을 적절히 수정하면 됨

3. 각 변수에 대한 설명

```
88          //3주차 실습
89          int dx[4] = { 1,-1,0,0 };
90          int dy[4] = { 0,0,1,-1 };
91
92          //3주차 과제
93          queue<pair<int, int>> q_bfs;
94
95          //void BFS(void);
96          void bfsdraw(void);
97          void init(void);
98          // 프로젝트
99          //dfs
00          void draw_DFS_PATH(void); //DFS 의 경로를 단계적으로 그려준다.
01          vector<pair<int, int>> path; // DFS 경로에 해당하는 점을 저장한다.
02          int dfs_print_idx = 0; // DFS경로를 출력하기위해 필요한 함수이다.
03          bool dfs_draw_finish = false; //DFS경로를 모두 출력했는지
04          //bfs
05          bool bfs_draw_finish = false; // BFS경로를 모두 출력했는지
06
07          //generate
08          void gen_maze(void); // maze 를 새로 만들기 위한 사전 작업
09          void g_maze(int x, int y); // maze를 만들때 사용되는 백트래킹 함수
10          int** new_maze; //새로 만든 maze를 저장하기 위한 포인터
```

변수의 선언은 다음과 같다. 이의 구체적인 역할은 4번 항목에서 확인할 수 있다.

4. 각 함수에 대한 설명

```
if (title == "Show DFS") {
    bShowInfo = bChecked;  // Flag is used elsewhere in Draw(
    if (isOpen)//DFS 를 실시한다.
    {
        isbfs = false;
        dfs_draw_finish = false;
        init(); // 기존에 프로그램을 실행한 결과들을 초기화시
        DFS(1, 1);
        bShowInfo = true;
        isdfs = true;
    }
    else
        cout << "you must open file first" << endl;
}

if (title == "Show BFS") {
    //doTopmost(bChecked); // Use the checked value directly
    if (isOpen)//BFS 를 실행한다.
    {
        isdfs = false;
        init();
        q_bfs.push(make_pair(1, 1));
        maze_arr[1][1] = 1;
        bShowInfo = true;
        isbfs = true;
    }
    else
        cout << "you must open file first" << endl;
}
```

먼저 DFS와 BFS 메뉴를 선택할 시에 이에 맞는 알고리즘을 실행한다. 하지만 DFS의 경우 클릭과 동시에 모든 작업을 수행하는 반면 BFS의 경우 큐에 시작값을 넣어주고 다음에 나오는 update함수에서 한 단계씩 이어나가준다.

```
void ofApp::update() {
    if (isdfs && !dfs_draw_finish) {
        dfs_print_idx++; // DFS 의 경로를 한단계씩 더 그려준다.
    }
    if (isbfs && !bfs_draw_finish) {
        //BFS의 단계는 DFS와는 다르게 update함수를 통해 진행한다.
        if (!q_bfs.empty()) {
            pair<int, int> curr = q_bfs.front();
            q_bfs.pop();
            int x = curr.first, y = curr.second;
            path.push_back(curr); // Store the current node to path
            if (x != WIDTH - 2 || y != HEIGHT - 2) {
                for (int i = 0; i < 4; i++) {
                    if (y + 2 * dy[i] < 0 || x + 2 * dx[i] < 0) continue;
                    if (maze_arr[y + dy[i]][x + dx[i]] == 1) continue;
                    if (maze_arr[y + 2 * dy[i]][x + 2 * dx[i]] != 0) continue;
                    maze_arr[y + 2 * dy[i]][x + 2 * dx[i]] = maze_arr[y][x] + 1;
                    q_bfs.push(make_pair(x + 2 * dx[i], y + 2 * dy[i]));
                }
            }
            else {//BFS가 종료된다면 큐에 있는 모든 원소를 빼준다.
                while (!q_bfs.empty()) q_bfs.pop();
                bfs_draw_finish = true;
            }
        }
    }
}
```

BFS 와 DFS의 과정을 단계별로 보여주어야 하기에 프레임이 바뀔때마다 호출되는 update 함수에서 한단계씩 진행한다. 차이점이 있다면 DFS의 경우 지나간 경로를 순차적으로 표시하기위해서 dfs_print_idx함수를 통해 한 단계씩 보여준다면 BFS는 각 장면별로 큐에 들어가 있는 값이 유효한 의미를 갖는다. 따라서 BFS를 모두 실행하고 부여주는 것이 아니라 프레임별 BFS를 다시 수행한다. BFS를 수행하면서 maze_arr에 저장되는 값은 심도이다. 기존 방문한방에 해당하는 값에서 1식 증가시켜서 배열에 할당해준다.

```
if (isbfs) {// BFS를 실행했다면
    ofSetColor(200);
    ofSetLineWidth(5);
    if (isOpen) {
        bfsdraw(); // 현재까지 bfs경로를 그려준다.
        if (!bfs_draw_finish) {
            pair<int, int> temp = q_bfs.front();
            // 현재 탐색중인 깊이에 있는 모든 점을 표한하기 위해
            // 첫 원소(지금 움직이는 원소)는 초록색으로 그려준다
            q_bfs.pop();
            ofSetColor(0, 255, 0);
            ofDrawCircle(temp.first * 22 + 2, temp.second * 22 + 2, 10);
            q_bfs.push(temp);
            //나머지 원소들은 하나씩 빼서 회색으로 그려주고 다시 큐에 넣어준다.
            for (int i = 1;i < q_bfs.size();i++) {
                temp = q_bfs.front();
                q_bfs.pop();
                ofSetColor(100);
                ofDrawCircle(temp.first * 22 + 2, temp.second * 22 + 2, 10);
                q_bfs.push(temp);
            }
            ofSetColor(100);
        }
    }
    else
        cout << "You must open file first" << endl;
}
```

다음은 BFS를 시각화 하기 위해서 작성된 코드이다. bfsdraw함수를 통해서 현재까지 진행된 BFS를 보여준다. 또한 BFS의 경우 큐에 들어있는 점의 정보를 표시하기위해서 큐의 가장 앞에있는 값(지금 움직이는 점)은 초록색, 다른 큐에 들어있는 점들은 하나씩 pop하여 출력하고 다시 push 하여 큐의 형태를 유지시켜준다.

```
if (isdfs)
{
    ofSetColor(200);
    ofSetLineWidth(5);
    if (isOpen) {
        if (!dfs_draw_finish)  //dfs가 완료되지 않았다면 중간 과정 까지 그려준다.
            draw_DFS_PATH();
        else
            dfsdraw(); // dfs가 완료되었다면 전체를 그려준다.
    }
    else
        cout << "You must open file first" << endl;
}
```

DFS의 경우 draw_DFS_PATH 를 통해 특정 위치까지의 실행 결과를 표시해 준다. Dfs_draw_finish 의 값이 true가 된다면 path의 모든 지점을 출력했다는 의미이기에 dfsdraw()함수를 통해 전체 탐색 정보를 출력한다.

```
bool ofApp::DFS(int x, int y)//DFS탐색을 하는 함수
{
    //TO DO
    //DFS탐색을 하는 함수 ( 3주차)
    //path 변수는 dfs에서 이동한 점들의 위치를 저장하는 변수이다.
    path.push_back(make_pair(x, y));
    if (x == WIDTH - 2 && y == HEIGHT - 2) {
        maze_arr[y][x] = 2;
        return true;
    }
    else {
        for (int i = 0;i < 4;i++) {
            if (x + dx[i] * 2 > 0 && x + dx[i] * 2 < WIDTH && y + dy[i] * 2 > 0 && y + dy[i] * 2 < HEIGHT && maze_arr[y + dy[i]][x + dx[i]] == 0) {
                if (maze_arr[y + 2 * dy[i]][x + 2 * dx[i]] == 0) {
                    maze_arr[y][x] = 3;;
                    maze_arr[y + 2 * dy[i]][x + 2 * dx[i]] = 3;
                    if (DFS(x + 2 * dx[i], y + 2 * dy[i])) {
                        maze_arr[y][x] = 2;
                        return true;
                    }
                    path.push_back(make_pair(x, y));
                    maze_arr[y][x] = 3;
                }
            }
        }
    }
    maze_arr[y][x] = 3;
    return false;
}
```

DFS함수에서는 실습과 동일하게 재귀호출을 통하여 maze_arr 값을 적절하게 설정해 준다. 이 프로젝트에서는 dfs의 경우에는 maze_arr에는 0,1,2,3 의 값이 저장된다. 올바른 경로 (시작지점에서 도착 지점으로 이어지는 경로) 는 2, 탐색하였지만 도착 지점으로 이어지지 않은 경로라면 3, 탐색하지 않은 경로는 0, 벽에 해당하는 위치에는 1이 저장된다. 이 함수에서 확인해야할 것은 path 벡터에 함수가 호출되었을 때와 호출에 실패하였을 때 모두 값이 들어간다는 것이다. 이렇게 작성한 이유는 dfs를 통해 막다른 길에 다다랐을 때 뒤로 돌아가는 형태를 표현하기 위함이다.

```
void ofApp::draw_DFS_PATH(void) { // path에 저장된 점을 기준으로 그려준다.
    if (dfs_print_idx >= path.size() - 1) {
        dfs_draw_finish = true;
        return;
    }
    for (int i = 0;i < dfs_print_idx;i++) { //path의 점을 이어서 그려준다.
        int x1 = path[i].first;
        int y1 = path[i].second;
        int x2 = path[i + 1].first;
        int y2 = path[i + 1].second;
        ofSetLineWidth(5);
        ofSetColor(200);
        ofDrawLine(x1 * 22 + 2, y1 * 22 + 2, x2 * 22 + 2, y2 * 22 + 2);
    }
    ofSetColor(100);
    ofDrawCircle(path[dfs_print_idx].first * 22 + 2, path[dfs_print_idx].second * 22 + 2, 10);
}
```

이 함수는 DFS 함수에서 저장한 정보를 담고있는 path의 값들을 이용하여 line과 점들을 그려준다. 이때 탐색을 진행하고 있으므로 모든 라인을 실패한 호출에 해당하는 색상으로 그려준다.

```
void ofApp::dfsdraw() // DFS가 완료되었다면 maze_arr의 정보를 기반으로 선을 그려준다.
{
    for (int i = 1;i < HEIGHT;i += 2) {
        for (int j = 1;j < WIDTH;j += 2) {
            for (int k = 0;k < 4;k++) {
                if (i + 2 * dy[k] < 0 || j + 2 * dx[k] < 0)
                    continue;
                if (maze_arr[i + dy[k]][j + dx[k]] == 1) continue;
                if (maze_arr[i][j] == 2 && maze_arr[i + 2 * dy[k]][j + 2 * dx[k]] == 2) {
                    ofSetColor(100);
                    ofDrawLine(j * 22 + 2, i * 22 + 2, (j + 2 * dx[k]) * 22 + 2, (i + 2 * dy[k]) * 22 + 2);
                }
                else if (maze_arr[i][j] == 2 && maze_arr[i + 2 * dy[k]][j + 2 * dx[k]] == 3) {
                    ofSetColor(200);
                    ofDrawLine(j * 22 + 2, i * 22 + 2, (j + 2 * dx[k]) * 22 + 2, (i + 2 * dy[k]) * 22 + 2);
                }
                else if (maze_arr[i][j] == 3 && maze_arr[i + 2 * dy[k]][j + 2 * dx[k]] == 2) {
                    ofSetColor(200);
                    ofDrawLine(j * 22 + 2, i * 22 + 2, (j + 2 * dx[k]) * 22 + 2, (i + 2 * dy[k]) * 22 + 2);
                }
                else if (maze_arr[i][j] == 3 && maze_arr[i + 2 * dy[k]][j + 2 * dx[k]] == 3) {
                    ofSetColor(200);
                    ofDrawLine(j * 22 + 2, i * 22 + 2, (j + 2 * dx[k]) * 22 + 2, (i + 2 * dy[k]) * 22 + 2);
                }
            }
        }
    }
}
```

dfsdraw의 경우 draw_dfs_path함수에서 모든 진행과정이 출력 되고 결과에 해당하는 화면을 출력할 때 사용된다. 이때는 성공한 길과 실패한 길을 구분해서 그려야 하기에 (2,2), (2.3), (3,3),(3,2)의 정보를 다르 색으로 구분하여 그려준다.

```
void ofApp::bfsdraw(void) { //dfs과정속 진행된 곳 까지 그려준다.
    for (int i = 1;i < HEIGHT;i += 2) {
        for (int j = 1;j < WIDTH;j += 2) {
            for (int k = 0;k < 4;k++) {
                if (i + 2 * dy[k] < 0 || j + 2 * dx[k] < 0)
                    continue;
                if (maze_arr[i + dy[k]][j + dx[k]] == 1) continue;
                if (maze_arr[i][j] != 0 && maze_arr[i + 2 * dy[k]][j + 2 * dx[k]] != 0) {
                    ofSetColor(200);
                    ofDrawLine(j * 22 + 2, i * 22 + 2, (j + 2 * dx[k]) * 22 + 2, (i + 2 * dy[k]) * 22 + 2);
                }
            }
        }
    }
    int x = WIDTH - 2, y = HEIGHT - 2;
    int depth = maze_arr[y][x];
    if (maze_arr[y][x] != 0) {
        while (depth != 1) {
            for (int k = 0;k < 4;k++) {
                if (maze_arr[y + dy[k]][x + dx[k]] == 1) continue;
                if (maze_arr[y + 2 * dy[k]][x + 2 * dx[k]] == depth - 1) {
                    ofSetColor(100);
                    ofDrawLine(x * 22 + 2, y * 22 + 2, (x + 2 * dx[k]) * 22 + 2, (y + 2 * dy[k]) * 22 + 2);
                    x = x + 2 * dx[k];
                    y = y + 2 * dy[k];
                    depth = maze_arr[y][x];
                    break;
                }
            }
        }
    }
}
```

bfsdraw함수의 경우 각 단계별로 실행된 bfs값을 표시하기 위해서 maze_arr에 값이 존재한다면 이를 출력한다. 만약 목표에 해당하는 값이 1이 아닌 값이 저장되어 있다면 이는 끝까지 탐색이 완료되었다는 의미이므로 올바른 경로를 찾아 색칠해준다.

```
void ofApp::gen_maze(void) {
    // maze를 생성해준다. 실습 시간과는 다르게 백트래킹을 통하여 구현하였다.
    // 만약 이미 파일이 열려있다면 메모리를 초기화해준다.
    if (isOpen) freeMemory();
    isbfs = isdfs = isOpen = false;
    int w, h;//w,h는 방의 수
    printf("input W,H : "); scanf("%d %d", &w, &h);
    //입력받은 크기에 맞게 동적 할당 해준다.
    WIDTH = 2 * w + 1; HEIGHT = 2 * h + 1;
    new_maze = (int**)malloc(sizeof(int*) * (2 * h + 1));
    for (int i = 0;i < (2 * h + 1);i++)
        new_maze[i] = (int*)malloc(sizeof(int) * (2 * w + 1));
    for (int i = 0;i < 2 * h + 1;i++)
        for (int j = 0;j < 2 * w + 1;j++)
            new_maze[i][j] = 1;
    //시작위치의 좌표를 랜덤함수를 통해 지정해준다.
    int sx = (rand() % w) * 2 + 1; int sy = (rand() % h) * 2 + 1;
    new_maze[sy][sx] = 0;
    g_maze(sx, sy); // g_maze라는 함수를 통해 백트래킹을 이용한 미로 생성을 실시한다.
```

Generate maze 메뉴를 선택하였을 때 실행되는 함수이다. 먼저 사용자로부터 만들 미로의 크기를 입력받는다. 그리고 이를 저장할 new_maze배열을 동적할당해

준다. 또한 백트랙킹을 통한 미로 생성을 할것이기에 모든 벽이 막혀 있도록 초기화해준다. 이후 미로를 형성할 때 시작되는 위치를 랜덤하게 설정하고 g_maze 함수를 호출하여 미로를 생성한다.

```
//만든 값을 저장하기위해서 input 과 maze_arr 배열에 동적할당 해준다.
input = (char**)malloc(sizeof(char*) * HEIGHT);
maze_arr = (int**)malloc(sizeof(int*) * (HEIGHT));
for (int i = 0;i < HEIGHT;i++) {
    input[i] = (char*)malloc(sizeof(char) * WIDTH);
    maze_arr[i] = (int*)malloc(sizeof(int) * (WIDTH));
}
// 생성한 미로를 input파일에 복사한다.
for (int i = 0;i < HEIGHT;i++) {
    for (int j = 0;j < WIDTH;j++) {
        if (i % 2 == 0) {
            if (j % 2 == 0)
                input[i][j] = '+';
            else {
                if (new_maze[i][j] == 1)
                    input[i][j] = '-';
                else
                    input[i][j] = ' ';
            }
        }
        else {
            if (new_maze[i][j] == 1)
                input[i][j] = '|';
            else
                input[i][j] = ' ';

        }
    }
}
init();
isOpen = 1;
```

미로 생성이 완료된다면 이를 input배열과 maze_arr배열에 저장해준다. 이를통해 우리는 파일을 저장하고, 화면에 표시할 수 있다.

```
//파일을 저장한다.
//ofFile을 사용한다면 기본 인코딩이 UTF-16으로 되기에 C언어의 FILE 을 사용한다.
char temp[20]; // 입력받는 메이즈 파일의 이름
char filename[30] = { "data/" }; // 폴더의 상대경로를 통하여 저장위치를 지정해준다.
printf("FILE NAME : ");
scanf("%s", temp);
strcat(filename, temp);
strcat(filename, ".maz"); // 확장자는 자동으로 .maz로 맞추어준다.
FILE* fp = fopen(filename, "w");
for (int i = 0;i < HEIGHT;i++) {
    for (int j = 0;j < WIDTH;j++)
        fprintf(fp, "%c", input[i][j]);
    fprintf(fp, "₩n");
}
fclose(fp);
// new_maze 는 사용이 완료되었으므로 메모리 할당을 해제해준다.
for (int i = 0;i < 2 * h + 1;i++)
    free(new_maze[i]);
free(new_maze);
```

파일을 저장하기위해서 ofFile확장자를 사용한다면 기본인코딩이 UTF-8이 아닌 UTF-16으로 되기에 우리가 원하는 형태의 파일을 만들 수 없다. 따라서 C언어의 FILE을 통하여 처리해주었다. Strcat을 통하여 입력된 파일의 이름의 상대 경로와 확장자를 설정하여 이후 프로그램을 실행할 때 해당하는 파일을 불러올 수 있도록 해준다. 작업이 완료된다면 new_maze의 경우 메모리 할당을 해제해준다.

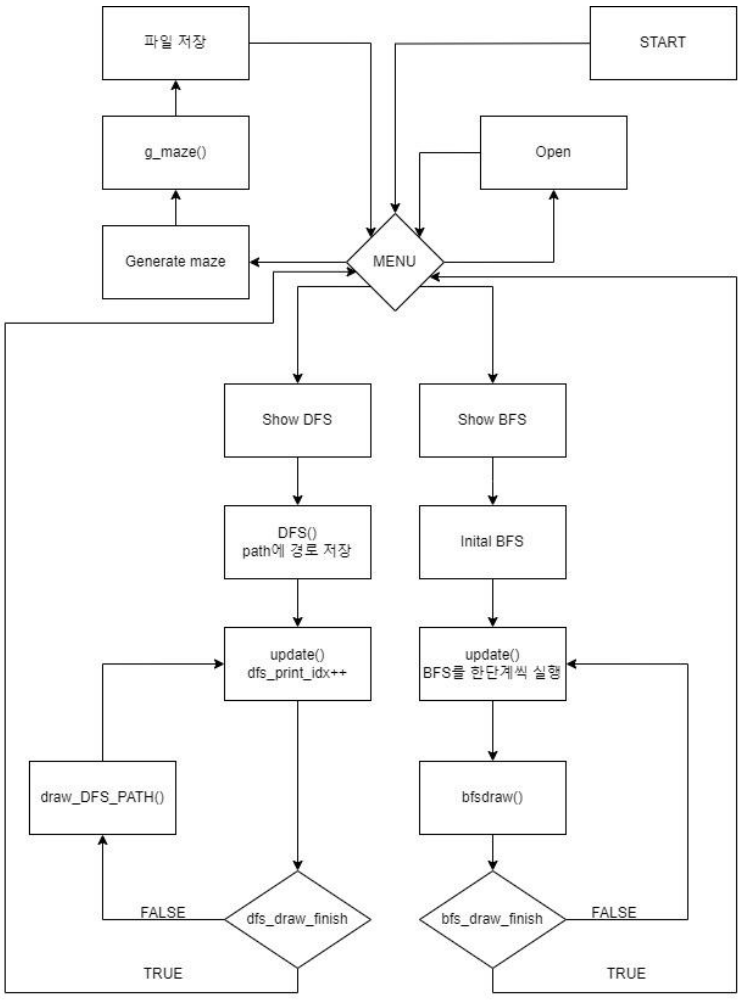
```
void ofApp::g_maze(int x, int y) {
    int d = rand() % 4; //랜덤한 방향을 지정해주고 그 방향의 벽을 허물어준다.
    for (int i = 0;i < 4;i++) {
        int tx = x + dx[(i + d) % 4] * 2;
        int ty = y + dy[(i + d) % 4] * 2;
        if (tx > 0 && tx < WIDTH && ty>0 && ty < HEIGHT && new_maze[ty][tx] == 1) {
            new_maze[ty][tx] = 0;
            new_maze[y + dy[(i + d) % 4]][x + dx[(i + d) % 4]] = 0;
            g_maze(tx, ty); // 재귀적으로 이를 반복한다.
        }
    }
}
```

g_maze 함수의 경우 백트래킹을 통하여 재귀적으로 미로를 생성한다. 이 알고리즘을 통하여 기존 실습에서 사용한 Eller의 알고리즘과 동일하게 완전 미로를 형성할 수 있다.

```
void ofApp::init(void) { // 기존 데이터 값을 지워주는 역할을 한다.
    for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
            if (input[i][j] != ' ')
                maze_arr[i][j] = 1;
            else
                maze_arr[i][j] = 0;
        }
    }
    path.clear();
    dfs_draw_finish = false;
    dfs_print_idx = 0;
    bfs_draw_finish = false;
}
```

다음은 init()함수이다. 이 함수는 BFS, DFS, 새로운 파일 호출, 미로 생성시 호출되면 기존에 설정된 flag역할을 하는 변수나 maze_arr등을 초기화 해주는 역할을 한다.

5. 전체 플로우차트



6. 자료구조

   먼저 .maz 파일을 읽고을때는 ofFile을 사용하였다. 하지만 ofFile을 이용하여 파일을 출력할경우 UTF-16 인코딩을사용한다. 이는 우리가 미로의 벽을 '+', '-', '|' 를 이용하여 출력하여 ASCII 코드를 기준으로 입력되어 UTF-8 로 인코딩하여야 유효한 의미를 가지는 파일을 볼 수 있기 때문에 출력에서는 FILE을 사용하였다. 또한 미로의 벽에 파일의 정보 input은 char 형 포인터를, 방문 여부를 저장하는 maze_arr 은 int 형 포인터를 사용하였다. DFS의 경우 중간 과정을 보여주기 위하여 중간에 저장하기위하여 C++ STL의 vector을 사용하였다. 특히 이때 x,y좌표를 저장하기위하여 vector<pair<int, int>> path; 와 같이 선언하였다. BFS경우 BFS 탐색에 필요한 C++ STL의 queue를 사용하였다. 이 또한 역시 queue<pair<int, int>> q_bfs를 이용하여 방문한 노드의 x,y좌표를 동시에 큐에 저장해 주었다.

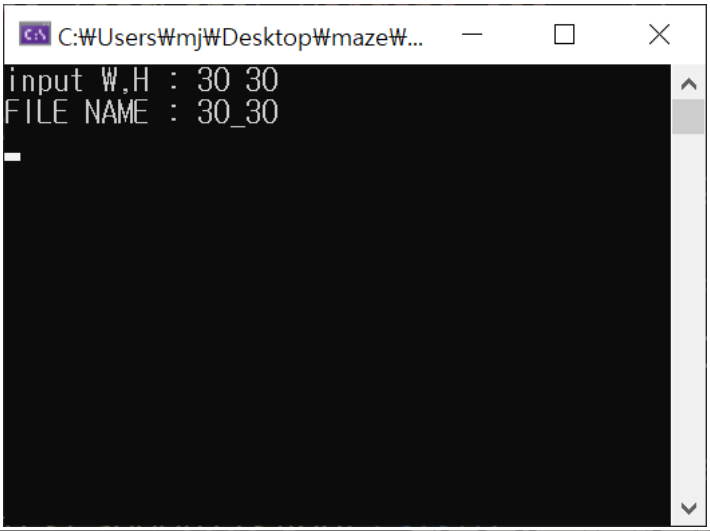
7. 시간, 공간 복잡도

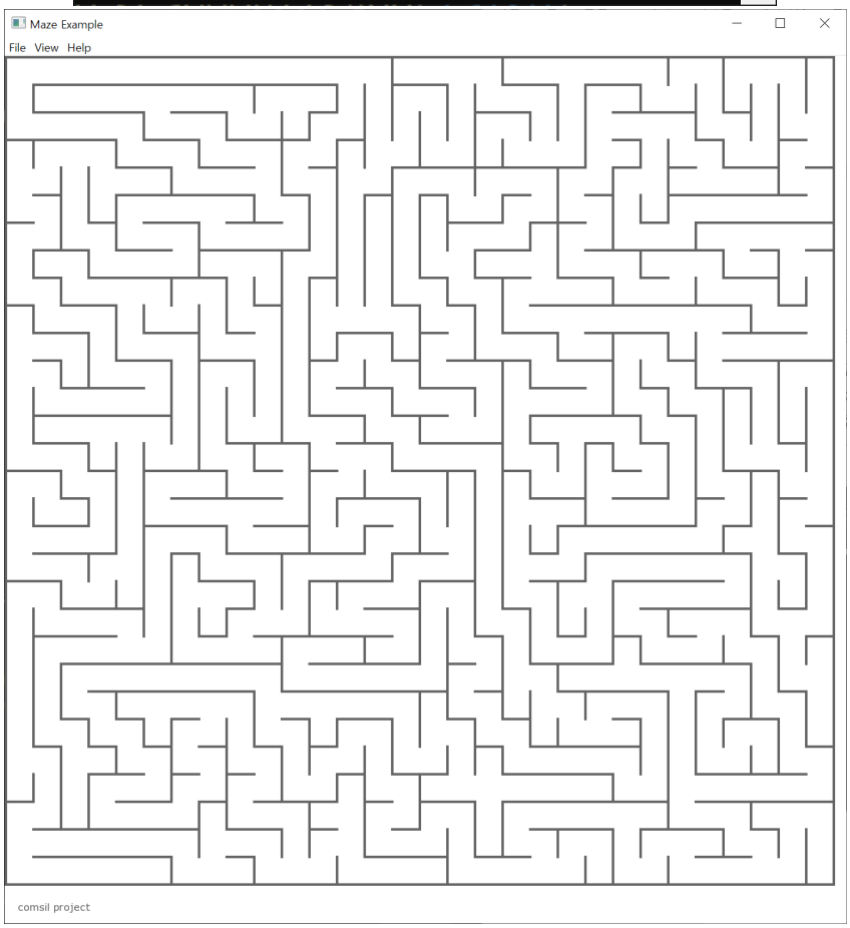
   먼저 기본적인 시간 복잡도는 모든 방을 방문하는 알고리즘이기에 O(W*H)이다. 이를 관리하기 위해서 BFS와 DFS모두 알고리즘을 한번만 사용하였다. 하지만 공간복잡도의 경우 DFS는 path에 실제 DFS 과정보다 더 많은 양의 정보를 저장한다. 이는 막다른 길에 도착하였을 때 뒤로 돌아가는 모습을 보여주기 위함이다. 그래서 최대 하나의 방이 path 벡터에 4 들어갈 수 있다. 하지만 asymptotically O(W*H)라고 볼 수 있다. BFS의 경우에는 시간복잡도와 공간복잡도가 모두 O(W*H)로 동일하다. 또한 백트래킹을 통한 미로의 생성 또한 한번 방문한 방은 다시 방문하지 않기 때문에 시간복잡도와 공간복잡도 모두 O(W*H)라고 볼 수 있다.

8. 창의적 구현

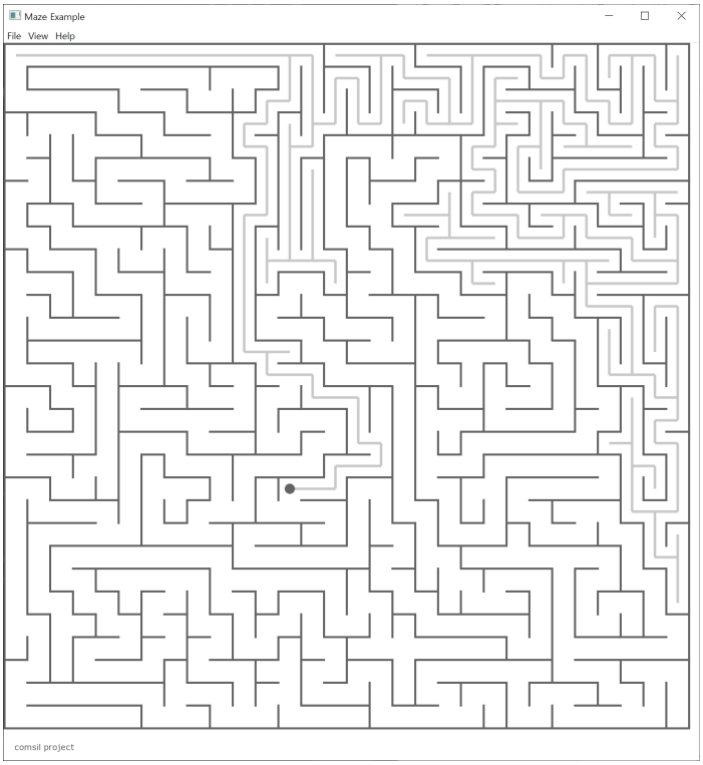
   Eller의 알고리즘이 아닌 recursive backtracking을 통하여 미로를 생성하였다. 또한 BFS의 시각화를 위하여 ofApp::update 부분에 BFS의 한단계에 해당하는 코드를 넣은 것 또한 창의적인 구현이었다고 생각한다. 또한 DFS는 이동하는 점을 시각화 하여 돌아가는 경로를 표시하는 것이 직관적인 설명이 되겠지만 BFS의 경우 같은 DEPTH의 방들을 탐색하기에 이를 돌아가는 형태로 구현하게 된다면 너무 조잡한 형태를 띠게되어 queue에 들어있는 방, 즉 앞으로 방문하게 될 노드의 경우를 다른색으로 화면에 표시하여 지금 어떤 방들이 탐색중인지 직관적으로 볼 수 있도록 하였다.

9. 프로젝트 실행 결과 화면

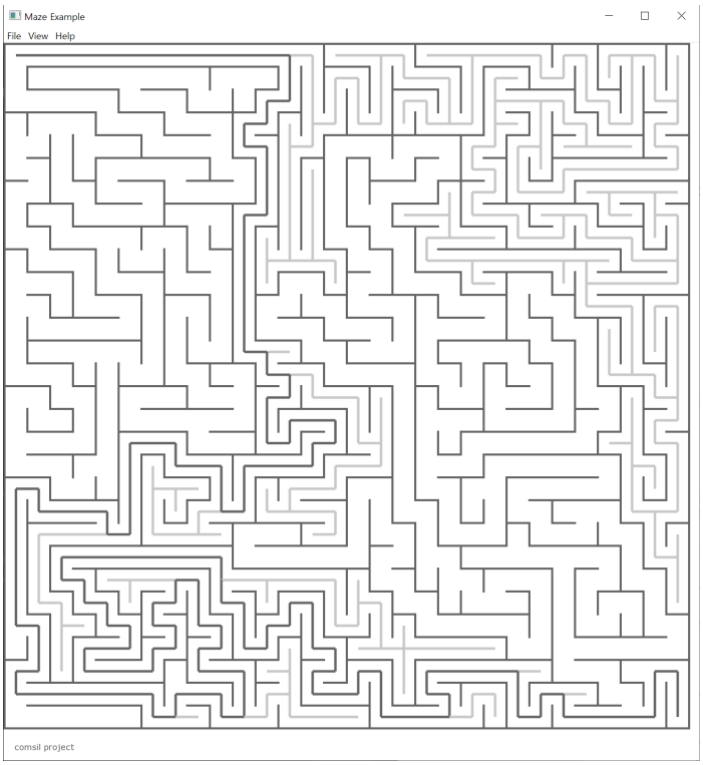




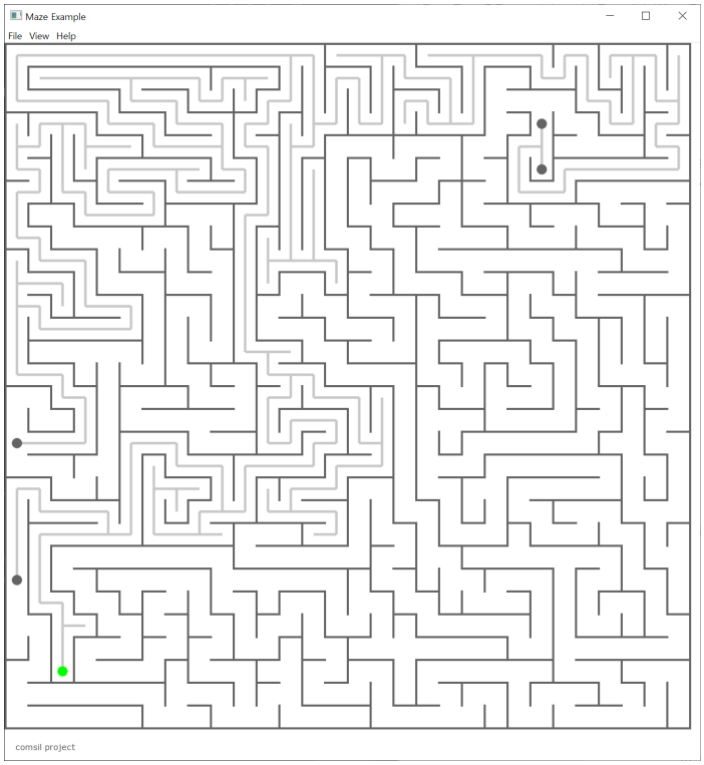
Generate maze메뉴를 통하여 30* 30 미로를 생성한 모습이다.



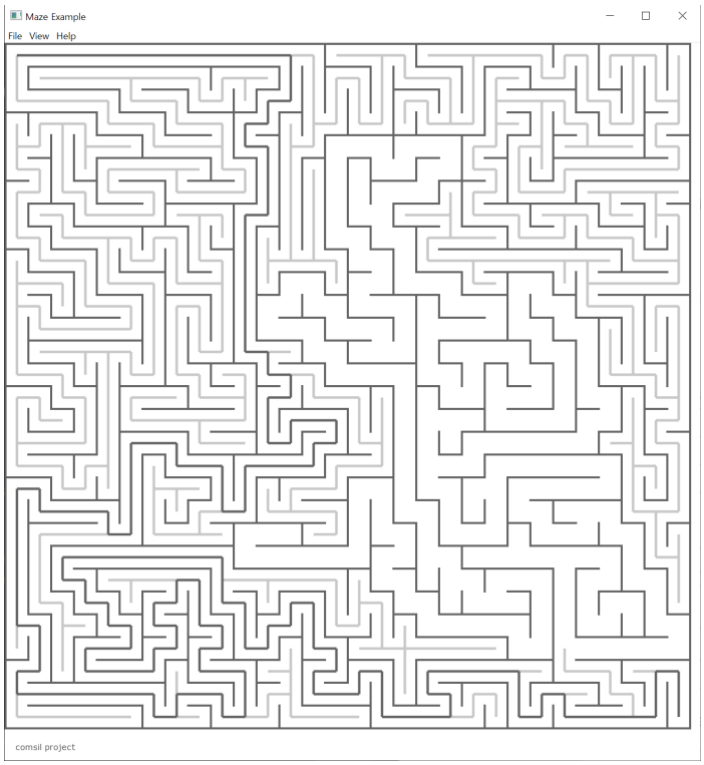
Show DFS 버튼을통해 DFS를 진행하고 있는 모습이다. 방문했던 모든 경로를 표시하고 있다.



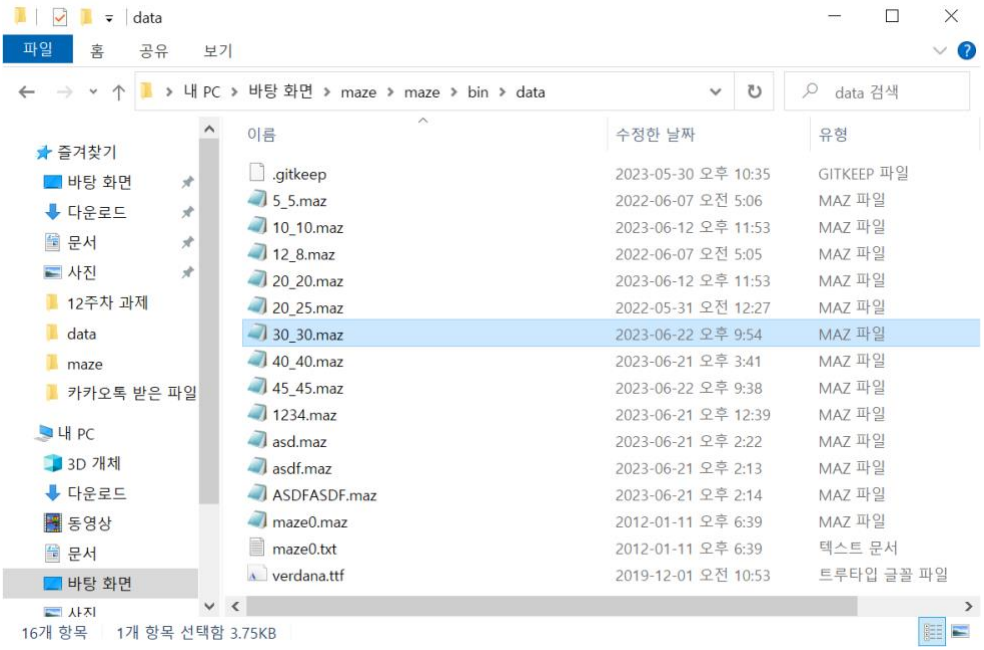
탐색이 완료된다면 올바른 경로는 진한 색으로 표시해준다.

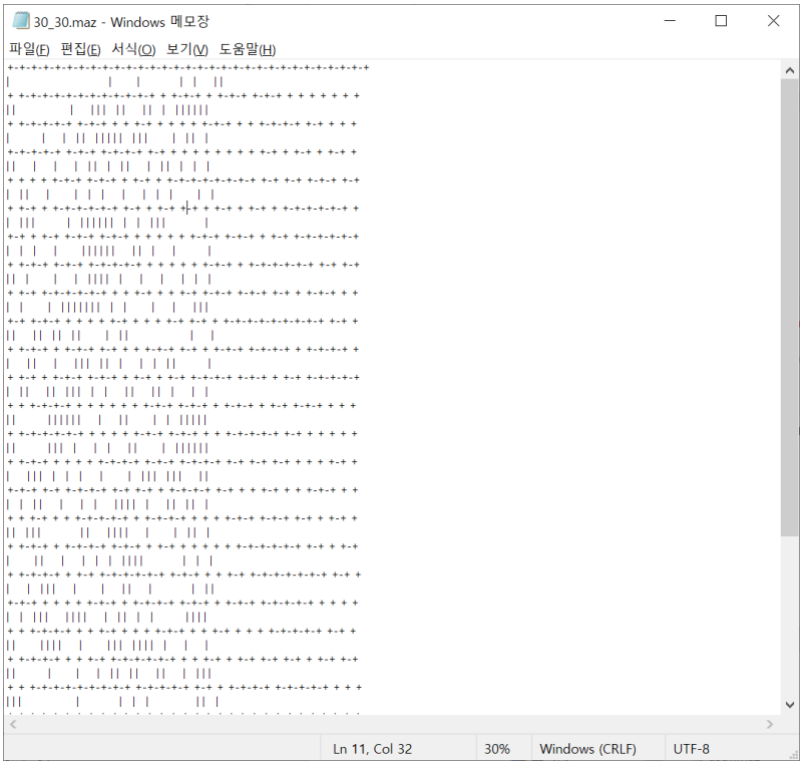


Show BFS를 통하여 탐색을 진행하고 있는 모습니다. 어떠한 방을 업데이트하는지 표시해 준다.



BFS탐색이 완료된 모습이다. DFS와 마찬가지로 올바른 경로를 진한색으로 표시해준다.





생성한 미로가 30_30.maz 의 이름으로 bin/data 에 저장되어 있음을 확인할 수 있다. 따라서 다음번 실행에도 기존에 생성한 파일을 다시 불러올 수 있다.

10. 느낀점 및 개선 사항

먼저 백트래킹을 통한 알고리즘 구현에 있어서 재귀적으로 호출되는 함수의 특성상 오픈 프레임워크 상에서 미로가 형성되는 과정을 시각화 하기 힘들다는 점점에 아쉬운점인 것 같다.