최종 프로젝트 보고서

전공: 철학과 학년: 3학년 학번: 20180032 이름: 남기동

1. 프로젝트 목표 및 실험 환경

1.1 프로젝트 목표

2022년 2학기 컴퓨터공학 설계 및 실험1의 최종 프로젝트는 미로 프로그램을 만드는 프로젝트이다. 미로 프로그램은 완전 미로를 생성하는 것, 완전 미로를 윈도우 화면에 그리는 것, 윈도우에서 완전 미로의 탈출 경로를 그리는 것 등으로 구성되었다. 2022년 2학기 컴퓨터공학 설계 및 실험1의 11주차부터 13주차까지 총 3주 동안 진행되었던 실습 내용을 바탕으로 새로운 기능을 추가하여 미로 프로그램을 전반적으로 개선하는 것을 프로젝트의 목표로 한다.

1.2 실험 환경

(1) 실험 기간

프로젝트는 2022년 11월 25일부터 12월 21일까지 약 1개월 동안 진행되었다. 11월 25일 1주차 실습을 통해 완전 미로 생성 프로그램을 작성하였으며 12월 2일 2주차 실습을 통해 Openframework에서 maz 확장자의 파일로부터 미로의 정보를 읽어서 윈도우에 출력하는 프로그램을 만들었다. 12월 9일 3주차 실습을 통해 DFS를 통한 미로 탈출 기능을 만들었고 이후 BFS를 통한 미로 탈출 기능을 추가하였다. 기말 시험 기간을 포함하여 최종적으로 12월 21일까지 기존 프로젝트에 기능을 개선하고 추가하여 프로젝트를 마무리하였다.

(2) OpenFramework

미로 프로젝트는 visual studio 2017을 기반으로 OpenFramework를 사용하여 구현되었다. OpenFramework는 C++를 기반으로 한 오픈 소스 라이브러리로서 "창의적인 코딩"을 위해 디자인되었다. 이번 프로젝트에서 OpenFramework를 사용하여 미로를 윈도우에 출력하고 미로와 관련된 시각적 작업을 처리하기 위해 활용하였다. OpenFramework는 ofApp.h와 ofApp.cpp를 기본 구조로 하며 여기에서 실질적인 코딩이 수행되었다. setup(), update(), draw() 함수가 기본적으로 설정되며 setup 함수는 응용 프로그램이 실행되는 순간에 한 번만 실행되며, 다양한 변수와 업데이트가 필요하지 않은 설정들을 선언하는 부분이다. update 함수는 매 프레임별로 지속적으로 작동한다. draw 함수는 화면에 매 프레임별로 그리고자 하는 작업을 포함한다.

(3) Windows.h, ofxWinMenu

앞서 이야기했듯 미로 프로젝트는 OpenFramework를 사용하여 구현되었으며 윈도우에서 미로와 관련된 작업들을 나타냈다. 윈도우를 사용하기 위해서는 Windows.h를 포함하였다. Windows.h는 윈도우 개발자들이 필요한 모든 매크로들, 다양한 함수들과 서브시스템에서 사용되는 모든 데이터 타입들 그리고 윈도우 API의 함수들을 위한 정의를 포함하는 윈도우의 C 및 C++ 헤더 파일이다. 여기에 더하여 ofxWinMenu.h와 ofxWinMenu.cpp를 추가하였는데 이것은 윈도우의 메뉴를 추가하는 역할로 사용된다. 이 부분들은 이번 미로 프로젝트 실질적으로 구현한 것이 아니라 미로와 관련하여 구현하고자 했던 바를 보여주기 위한 환경을 제공하기 위해 활용한 부분이다. 따라서 이 부분에 관련한 부분보다는 각 주차별로 구현한 미로의 기능과 최종 프로젝트의 완성을 위해 새롭게 구현한 내용들 위주로 보고서를 작성하였다.

2. 프로그램 전체 개요

2.1 자료구조

(1) 그래프

기본적으로 미로의 방을 나타낼 때 사용한 자료구조이다. 노드는 row, col, index, middle_x, middle_y, node_pointer link[4]로 구성되어 있다. 각각의 기능은 다음과 같다.

row, col: readfile함수에서 maz확장자 파일의 정보를 읽어서 adjacency matrix 형태로 정보를 저장하게 된다. 이 때 matrix의 열과 행을 나타내는 값으로 row와 col을 사용하게 된다.

index: index는 미로의 방을 의미하는 여러 노드가 있을 때 해당하는 노드가 몇 번째 방인지를 나타낸다. 좌측 상단에서부터 0으로 인식하여 한 행의 끝에 도달하면 다음 행의 첫 번째 열을 다음 인덱스로 간주한다.

middle_x, middle_y: 3주차 실습과 과제의 DFS, BFS를 사용하여 화면에 나타내는 과정에서 필요한 값이다. 미로의 방의 네 꼭짓점이 있다고 할 때 네 꼭짓점의 대각선이 교차하는, 즉 미로의 방의 중심부의 좌표를 나타낸다.

node_pointer link[4]: adjacency list로 adjacency matrix를 변환하였을 때 해당 미로의 방을 기준으로 네 방향의 방을 가리킬 수 있는 포인터 변수이다. 상하좌우, 총 4가지 방향이 있기 때문에 사이즈 4의 배열로 되어있으며 link[0]은 위쪽을, link[1]은 오른쪽을, link[2]는 아래쪽을, link[3]은 왼쪽을 의미한다.

이렇게 정의한 노드(node)를 활용하여 미로의 방의 정보를 저장하는 "node_pointer* graph"에 사용한다. Graph는 미로의 너비와 높이가 주어졌을 때 두 값의 곱으로 총 미로 방의 개수를 구하고 그 크기만큼 동적 할당하여 사용한다.

(2) 스택

스택은 3주차 실습과 과제에서 구현하는 DFS, BFS 및 그 결과물을 윈도우 화면에 출력하는 과정에서 사용되는 자료구조이다. 특히 DFS의 기능 자체를 스택을 기반으로 구현했기 때문에 DFS 함수에서 주로 사용된다.

스택은 범용적으로 사용되는 스택의 사용법을 그대로 사용하였으며 push와 pop 기능을 포함하고 있다. 이번 프로젝트에서 사용된 스택의 유일한 다른 점은 push함수에서 int* stack에 top을 활용하여 값을 추가하는 것 이외에 int* all에도 값을 추가한다. Int* all은 이후 DFS를 화면에 그리기 위해 방문했던 곳의 정보까지도 포함하기 위한 용도로 사용된다.

(3) 큐

스택은 3주차 실습과 과제에서 구현하는 DFS, BFS 및 그 결과물을 윈도우 화면에 출력하는 과정에서 주로 사용되는 자료구조이다. 특히 BFS의 기능 자체를 큐를 기반으로 구현했기 때문에 BFS 함수에서 주로 사용된다.

큐 역시 스택과 동일하게 범용적으로 사용되는 스택의 사용법을 그대로 사용하였으며 add와 delete 기능을 포함하고 있다. 이번 프로젝트에서 사용된 스택의 유일한 다른 점은 add함수에서 int* stack에 rear을 활용하여 값을 추가하는 것 이외에 int* all에도 값을 추가한다. Int* all은 이후 BFS를 화면에 그리기 위해 방문했던 곳의 정보까지도 포함하기 위한 용도로 사용된다.

2.2 멤버 변수와 멤버 함수 개괄

(1) 1주차 실습

void make_maze(): 아래 함수들을 포함하여 완전 미로를 만들고 maze.maz 파일 안에 저장

void Init_maze(int temp_height, int temp_width, int** maze): 2차원 배열의 미로를 초기화

void print_maze(int temp_height, int temp_width, int** maze): 디버깅 목적으로 미로의 결과물을 콘솔에 출력

void fwrite_maze(int temp_height, int temp_width, int** maze): 미로의 결과물을 maze.maz 이름의 파일을 생성하고 저장

void Eller_algorithm(int temp_height, int temp_width, int** maze): 엘러의 알고리즘으로 완전 미로를 생성하는 알고리즘

(2) 2주차 실습

bool readFile(): maz확장자 파일을 읽어서 미로의 정보를 저장해오는 역할을 하는 함수

void freeMemory(): 동적 할당했던 메모리들을 해제하는 역할을 하는 함수

<adjacency matrix로 구현한 미로>

int** maze: 파일의 미로 정보를 담는 이차원 배열

int HEIGHT: 미로의 높이

int WIDTH: 미로의 너비

<adjacency list로 구현한 미로>

node_pointer* graph: graph[MAX_VERTICES]로 동적 할당된다

int height_graph: 미로 방 기준, 미로의 높이

int width_graph: 미로 방 기준, 미로의 너비

int num_of_terms: 미로 방의 개수

(3) 3주차 실습

bool DFS(): DFS 경로 탐색을 활용하여 도착점까지의 너비 우선 탐색의 결과를 도출하는 함수

void dfsdraw(): DFS가 성공적으로 완료되어 경로가 나왔을 때 이를 화면에 그리는 함수

int* visited: 각 노드(미로의 방)에 방문했는지를 표시하기 위한 일차원 배열

int* stack: DFS구현을 위한 자료구조로 num_of_terms만큼 동적 할당해서 경로 저장 용도로 활용

int top: stack의 구현을 위해 필요한 변수

int* all: DFS함수에서 방문했던 모든 경로를 저장하기 위한 용도의 일차원 배열

int top_all: all을 stack으로 구현할 때 이용하는 변수

void push(int current_pos): stack의 기본 기능으로 현재 위치를 입력 받아 스택에 넣는다

int pop(): stack의 기본 기능으로 스택의 top에 해당하는 값을 반환한다

bool BFS(): BFS 경로 탐색을 활용하여 도착점까지의 너비 우선 탐색의 결과를 도출하는 함수

void bfsdraw(): BFS가 성공적으로 완료되어 경로가 나왔을 때 이를 화면에 그리는 함수

int* queue: BFS를 구현하기 위해 사용하는 queue 자료구조

void addq(int current_pos): 현재 위치를 입력으로 받아서 rear을 증가시켜 queue에 하나 더한다

int deleteq(): queue의 기본적인 기능 중 하나로 front를 증가시켜 queue를 하나 제거한다

int front, rear: queue의 구현을 위해 필요한 front와 rear

int rear_all: all에 queue로 저장하기 위해 사용하는 rear 역할의 변수

(4) 그 외

int isOpen: 파일이 열렸는지를 판단하는 변수. 0이면 안 열렸고 1이면 열렸다.

int isDFS: DFS함수를 실행시켰는지 판단하는 변수. 0이면 실행 안 했고 1이면 실행했다.

int isBFS: BFS함수를 실행시켰는지 판단하는 변수. 0이면 실행 안 했고 1이면 실행했다.

(5) 추가 및 개선 작업

ofImage Jerry: 제리의 이미지를 저장하는 변수

ofImage Cheese: 치즈의 이미지를 저장하는 변수

int entrance: 미로의 시작점을 의미

int exit: 미로의 도착점을 의미

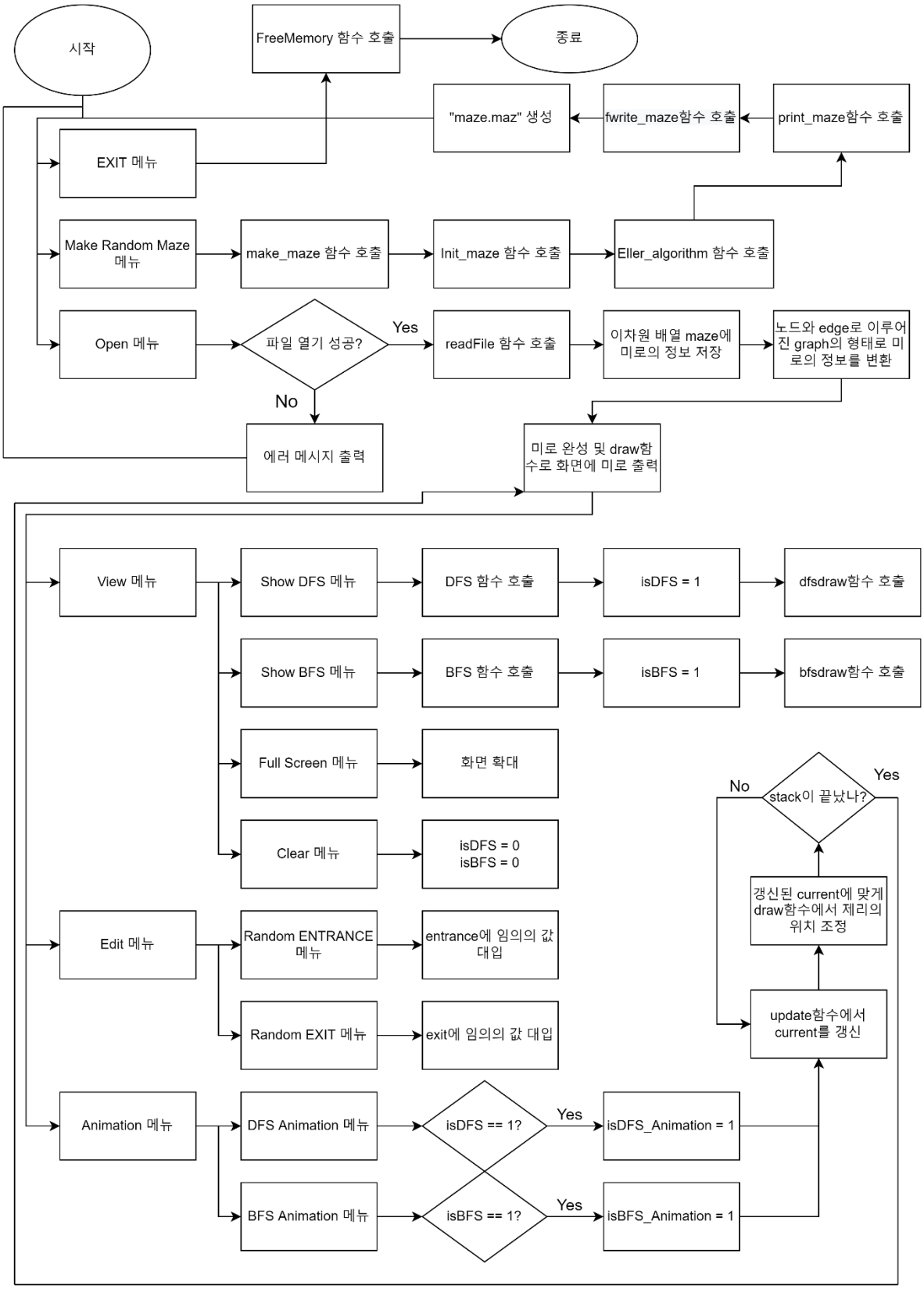
int current_top: stack의 top과 같은 역할을 하는 변수

int current: stack[current_top]을 대입해서 현재 위치해야 하는 미로의 방을 가리키는 index

int isDFS_Animation: 1이면 DFS_Animation을 실행시키도록 하는 flag

int isBFS_Animation: 1이면 BFS_Animation을 실행시키도록 하는 flag

2.3 플로우 차트(Flow Chart)



3. 각 주차별 구현 내용

3.1 미로 프로젝트 1주차

미로 프로젝트 1주차 실습에서는 완전 미로(Perfect Maze)를 생성하는 알고리즘을 작성하였다. 완전 미로는 미로에서 임의로 서로 다른 출발점과 도착점을 설정할 때, 두 지점을 연결하는 경로가 오로지 하나 존재하는 미로를 의미한다. 즉, 폐쇄된 공간이나 순환 경로가 존재하지 않는 미로인 것이다. 이와 반대되는 개념으로는 불완전 미로(Imperfect Maze)가 있다. 불완전 미로는 폐쇄된 공간이나 순환 경로가 존재하여 임의의 두 지점을 연결하는 경로가 하나 이상 존재하는 미로이다.

완전 미로를 생성하기 위해서 1주차 실습에서 사용한 알고리즘은 엘러의 알고리즘(Eller's Algorithm)이다. 엘러의 알고리즘은 첫 번째 행에서부터 시작해서 하나의 행을 완성하면서 아래로 나아가는 미로 생성 방식이다. 한번 행을 만들고 나면 다시 행을 조사할 필요가 없기 때문에 시간복잡도는 미로의 크기를 n이라고 했을 때 O(n)의 선형 시간을 갖게 된다.

구체적인 알고리즘은 다음과 같다.

1. 미로의 첫 번째 줄을 초기화한다. 첫 번째 줄에 속한 N개의 방은 모두 다른 집합에 속한다
2. 미로의 현재 행에서 첫 번째 방부터 시작해서 마지막 방까지 서로 인접해 있는 방들이 어떤 집합에 속하는지 비교한다. 인접해 있는 두 방이 서로 다른 집합에 속해 있다면, 두 방 사이의 벽을 남겨둘지 제거할지 임의로 선택한다. 벽을 제거하면 인접한 방들을 연결하는 통로가 생기는데 이렇게 해서 연결된 방들은 모두 같은 집합으로 합쳐진다.
3. 현재 줄과 다음 줄 사이의 벽을 제거하고 두 방 사이의 수직 경로를 만든다. 현재 줄의 방 중 하단의 벽을 제거하고 다음 줄의 방과 연결될 방은 각 집합마다 하나 이상을 임의로 선택하여야 한다. 두 방 사이에 수직 경로가 만들어지면, 이러한 두 방이 같은 집합에 속하게 되므로 첫 번째 줄의 방과 같은 집합에 속한 방들이 두 번째 줄에 나타나게 된다.
4. 3번에서 완성하지 않은 다음 줄의 나머지 방들을 구체화한다. 바로 전 줄과 수직의 경로로 연결된 방들 이외의 나머지 방들은 각각 서로 다른 집합에 속하게 된다.
5. 마지막 줄에 도달할 때까지 2~4번 과정을 반복한다
6. 미로의 마지막 줄에선 인접해 있으며 서로 다른 집합에 속한 방들 사이의 모든 벽을 제거한다.

입력으로는 미로의 너비와 미로의 높이를 받게 되고 출력으로 입력된 너비와 높이를 갖는 완전 미로를 maz를 확장자로 갖는 파일에 출력해야 한다. 이 때 각 방의 모서리는 '+'로, 가로벽은 '-', 세로벽은 '|', 그리고 방은 ' '과 같이 빈칸으로 나타낸다. 이러한 요구사항을 충족시키는 프로그램을 작성하기 위해서는 엘러 알고리즘에 기반하면서도 실습 환경과 요구에 맞추어 실제 구현은 구

체화되고 변형되어야 한다.

실습 환경과 요구 사항에 맞춘 알고리즘 구현은 다음의 절차를 따른다.

1. 미로의 너비와 높이를 입력 받는다

2. 미로의 너비*2 +1, 미로의 높이*2 +1 만큼의 크기를 갖는 미로를 이차원 배열에 동적 할당한다

3. 미로를 만들기 전 필요한 사전 작업을 하여 초기화를 한다(Init maze 함수 작성)

(행과 열 모두 짝수인 경우는 모서리('+')/ 행이 짝수, 열이 홀수인 경우는 가로벽('-')/ 행이 홀수, 열이 짝수인 경우는 세로벽('|')/ 행과 열 모두 홀수인 경우는 미로의 셀 혹은 방(' ')으로 간주한다)

4. 첫 번째 줄이 서로 다른 집합에 속하도록 WIDTH만큼 탐색하며 미로의 셀에 number를 증가시키면서 대입시킨다

5. 첫 번째 줄의 벽을 랜덤으로 제거 혹은 유지하며 제거되어 연결된 셀의 경우에는 같은 집합으로 파악하도록 같은 값을 대입한다

6. 이후부터는 마지막 줄 전까지 아래 사항을 준수하며 동일한 수행을 반복한다

    1) 수직 경로를 만드는데 무조건 집합 당 1개의 수직 경로는 있어야 하며, 이미 집합에서 수직 경로가 하나 있다면 이후의 셀들은 임의로 벽을 제거하거나 유지하게 한다.

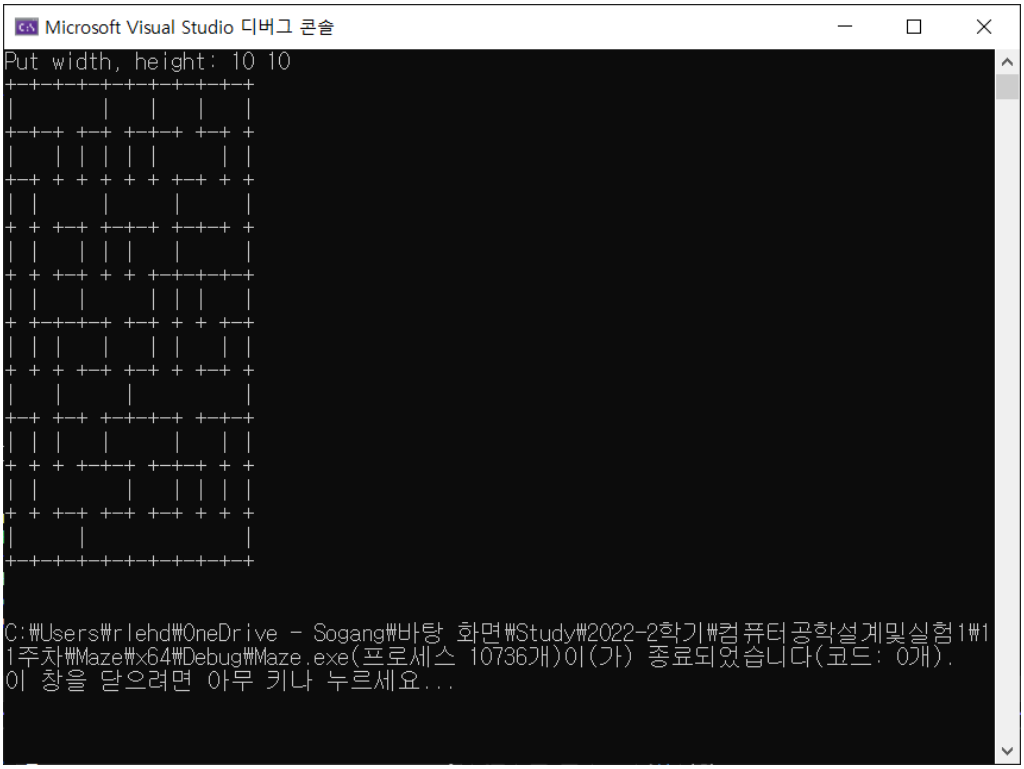
    2) 아래 셀을 읽어서 0으로 초기화된 값이 안 바뀌었으면, 수직 경로가 없다는 의미이고 이 경우에는 새로운 집합을 만들며, 같은 행에서 임의로 벽을 제거한다

7. 마지막 줄은 인접해 있으면서 서로 다른 집합에 속한 방들 사이의 모든 벽을 제거한다.

8. maze, HEIGHT, WIDTH를 입력 받아 화면에 출력하는 print_maze 함수를 만든다. (디버깅용)

9. print_maze 함수를 변형하여 'maze.maz' 파일에 미로를 적는 fwrite 함수를 만든다.

10. 동적으로 할당했던 값들을 해제한다.



<실습 1주차 이후 출력 결과>

미로 1주차 실습에서 만든 완전 미로 생성 프로그램은 2주차, 3주차 실습의 환경과는 별개로 구현되었다. 2~3주차 실습에서는 완전 미로를 읽고 미로의 정보를 저장하며 이를 화면에 나타내거나 DFS, BFS와 같은 탐색 방법으로 미로를 탈출하는 경로를 윈도우에 나타냈다. 이번 미로 프로젝트를 하나의 프로그램으로 단일하게 만드는 것이 프로젝트의 완성도를 높일 것이라 기대되기 때문에 1주차 실습의 완전 미로 생성 프로그램을 Openframeworks 작업 환경에 맞게 변형하여 윈도우 내에서 미로를 만들 수 있도록 설정하였다.

3.2 미로 프로젝트 2주차

2주차 실습에서는 총 3가지 함수를 구현했다. 첫 번째는 readFile 함수이고 maz 확장자의 미로 정보를 갖고 있는 파일을 읽어서 파일의 정보를 토대로 maze를 만드는 함수이다. 두 번째는 freeMemory함수이고 readFile에서 읽은 정보를 토대로 미로의 정보를 저장하기 위해 동적 할당하여 사용했던 변수인 maze와 graph의 메모리를 해제하는 역할을 수행한다. 마지막은 draw함수로 draw함수는 OpenFramework의 기본 함수 중 하나인데 프레임별로 draw 함수 내의 작업을 윈도우 화면에 그리는 역할을 수행한다. freeMemory 함수와 draw함수는 이후 3주차 프로젝트, 나아가 최종 프로젝트를 위한 기능 추가 및 개선에서 추가되었다. 각 함수에 대해 세부적인 알고리즘과

시공간 복잡도를 설명하면 다음과 같다.

(1) readFile 함수

미로의 정보를 저장하기 위해 선택한 자료구조는 그래프이다. 그래프는 vertex와 edge로 이루어져 있는 자료구조로 vertex를 노드로 생각하고 edge를 노드의 link로 생각하여 자료구조를 설계했다.

이때 link는 일차원 배열로 0부터 4까지 가능한데 0은 위쪽, 1은 오른쪽, 2는 아래쪽, 3은 왼쪽으로 이동할 수 있는 길이 있는지를 의미하며 벽이 뚫려 있어서 길이 있다면 연결되어 있는 방을 가리키는 노드를 포인터로 가리키게 한다.

그러나 이 자료구조를 구성하기 이전에 파일의 정보를 담는 maze를 따로 사용했는데, maze는 int형 2차원배열이다. 각각의 term은 정수를 갖고 있는데 1은 미로에서 모서리('+')를 의미하고 2는 가로벽(-), 3은 세로벽(|), 4는 가로벽이나 세로벽이 없어진 자리, 0은 미로의 방을 의미한다. Maze로 변환하여 받아온 파일의 정보를 graph의 node로 변환하여 저장하는 방식을 사용했다.

실질적인 알고리즘은 파일을 읽어서 미로의 높이와 너비를 구한 다음 HEIGHT와 WIDTH에 저장하는 것에서 시작한다. 그리고 2차원 배열 maze에 미로의 정보를 넣을 것이므로 높이와 너비를 참고하여 메모리를 동적 할당한다

다음으로, 파일의 내용을 읽어서 maze에 저장한다. 이때, 모서리는 1로, 가로벽은 2로, 세로벽은 3으로, 그리고 세로벽과 가로벽들이 있어야 할 위치인데 빈칸인 경우에는 세로벽과 가로벽이 없어진 것을 의미하므로 이를 나타내기 위해 4를 대입하고 나머지는 미로의 방을 나타내기 위해 0을 대입한다.

그 다음으로 graph를 만들기 위해 선작업이 필요하다. Graph의 index를 가리킬 number를 0으로 선언하고 height_graph와 width_graph는 각각 HEIGHT와 WIDTH를 2로 나눈 몫을 저장하는데, 이것은 실질적으로 미로의 방만을 따졌을 때의 미로의 높이와 미로의 너비를 의미한다. 그리고 미로의 방의 개수를 num_of_terms로 정의하고 graph에 node_pointer 자료형 크기를 num_of_terms 만큼 곱하여 메모리를 할당한다.

이어서 maze를 읽어서 새로운 자료구조인 graph의 모든 node를 구성하고 초기화한다. Node는 미로에서의 row와 col 값을 갖고 있으며 자신이 몇 번째 노드인지를 나타내는 index와 link[4]를 갖는다. Link[4]는 위, 오른쪽, 아래, 왼쪽에 해당하는 edge를 상징한다. 초기화이기 때문에 link[4]의 값들은 모두 NULL로 저장한다. 왜냐하면 위에서부터 차례대로 노드를 읽어서 만드는데 아직 못 만든 노드를 가리키는 경우도 있을 수 있기 때문이다. 이후 maze의 정보를 graph로 변환한

이후에 각각의 노드를 읽을 때 link의 값이 NULL인 것은 해당 방향으로 벽이 만들어져 있음을 의미한다.

마지막으로 초기화된 graph의 노드들을 차례대로 읽어가면서 열려 있는 방향에 해당하는 노드와 edge를 만들어준다.

readFile함수의 시간 복잡도는 미로의 높이를 HEIGHT, 미로의 너비를 WIDTH라고 할 때, O(HEIGHT * WIDTH)이다. 왜냐하면 파일의 정보를 읽어서 maze라는 이차원 배열에 저장하는 것이 HEIGHT와 WIDTH를 이중 루프로 돌기 때문이다. 이후 maze를 이용하여 graph를 만들 때도 HEIGHT와 WIDTH를 이중 루프로 돌아 시간 복잡도가 동일하다. 공간 복잡도는 maze와 graph 이 두 개를 동적 할당하여 메모리의 크기를 따져봐야 하는데, maze의 크기는 HEIGHT*WIDTH이며 graph의 크기는 height_graph*width_graph이다. 따라서 공간 복잡도도 O(HEIGHT * WIDTH)이다.

(2) freeMemory 함수

이차원 배열로 할당했던 maze를 먼저 해제해주고 node_pointer의 자료형을 갖고 있으며 1차원 배열인 graph도 해제해준다.

이후에 추가된 것은 만약 isDFS가 1이라면, DFS에 사용되기 위해 동적 할당되었던 stack, visited, all을 메모리 해제해주고, isBFS가 1이라면, BFS를 사용하기 위해 동적 할당되었던 queue, visited, all, stack을 메모리 해제해주어야 한다.

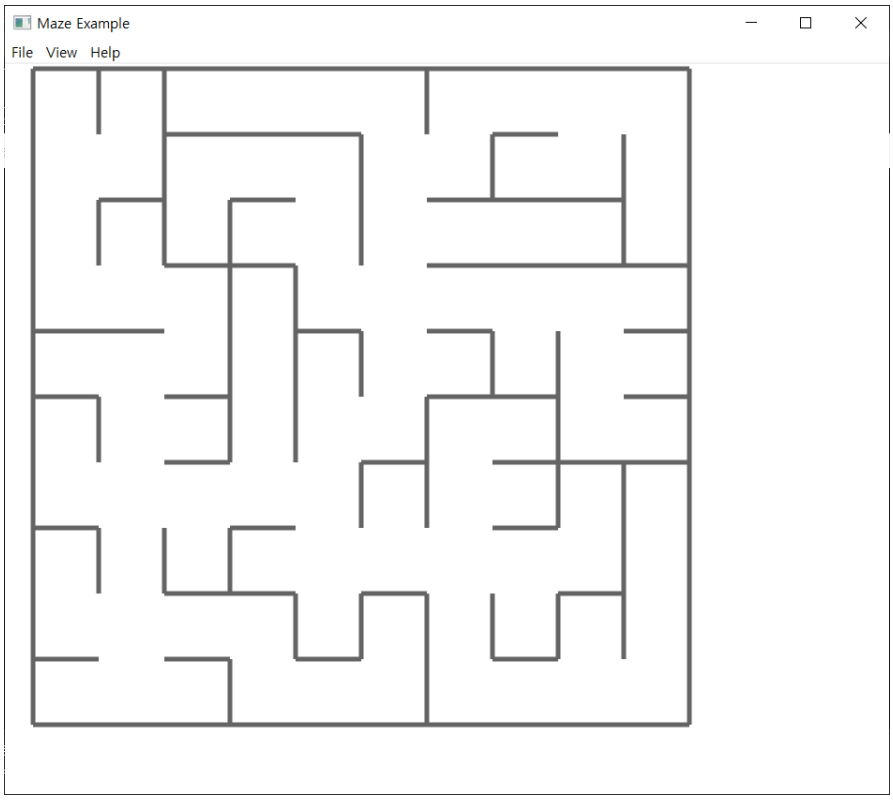
(3) draw 함수

파일을 읽고 graph의 노드를 만든 것을 토대로 draw함수를 사용하여 화면에 미로를 그려야 한다. draw함수는 파일을 읽으면서 새롭게 정의한 graph의 노드들을 하나씩 읽어가면서 노드의 4방향에 해당하는 link 값이 NULL이면 다른 노드와 연결되어 있지 않다는 것을 의미하기 때문에 벽을 그려주고 그렇지 않다면 빈 공간으로 남겨두는 방식을 사용하였다.

x, y는 미로를 그릴 때의 시작점이며 line_length는 한 노드를 기준으로 벽을 그릴 때 벽의 길이를 의미한다. 그리고 파일이 열려 있는 경우에만 graph의 노드들이 생성되기 때문에 if(isOpen)을 사용해야 한다. 미로의 높이와 너비를 이중 루프로 돌면서 노드를 차례대로 조사하여 위, 오른쪽, 아래, 왼쪽에 해당하는 방향의 link 값이 NULL인 경우에 벽을 그려준다. 연결된 노드가 없다면 벽인 것을 의미하기 때문에 그런 경우에만 ofDrawLine을 사용하여 벽을 그리는 것이다. 하나의 노드를 다 그렸으면 다음 노드를 그리기 위한 꼭짓점으로 이동하기 위해 x의 값에 line_length를 더하고 graph의 다음 node를 가리키기 위한 인덱스 역할을 하는 number도 1 증가시킨다. 그러다

가 한 줄을 다 그렸으면 처음의 x 값으로 돌아오고 y 값은 그 때 line_length만큼 증가시켜서 한 줄 아래 맨 왼쪽에서부터 다시 노드를 조사하고 그릴 수 있도록 한다.

2주차 실습 완료 후 draw함수의 시간 복잡도는 O(HEIGHT * WIDTH)이다. graph의 정보를 처리하고 draw함수에서 graph의 정보를 읽어서 화면에 출력하는 과정은 height_graph와 width_graph를 이중 루프로 도는데 이 값들도 HEIGHT와 WIDTH를 2로 나눈 몫이기 때문에 화면에 미로를 그리기 위한 시간 복잡도는 O(HEIGHT * WIDTH)이다. 공간 복잡도에 관해서는 readFile에서 만든 graph를 사용하지만 draw 함수에서는 따로 동적 할당한 것 없이 지역 변수만을 활용하여 구현하였다.



<2주차 실습 이후 출력 결과>

3.3 미로 프로젝트 3주차

3주차 실습 및 과제에서는 DFS와 BFS로 좌측 상단을 출발점으로 우측 하단을 탈출구로 가정했을 때의 경로를 탐색하는 기능을 구현했다. 실질적으로 DFS와 BFS의 알고리즘, 시공간 복잡도를 보기 전에 DFS와 BFS가 무엇이고 어떠한 차이점이 있는지, 어떤 자료 구조를 활용하여 구현할 것인지를 파악해볼 필요가 있다.

DFS는 Depth First Search의 줄임말로 깊이 우선 탐색 방식이다. DFS는 시작점에서 출발하여 가능

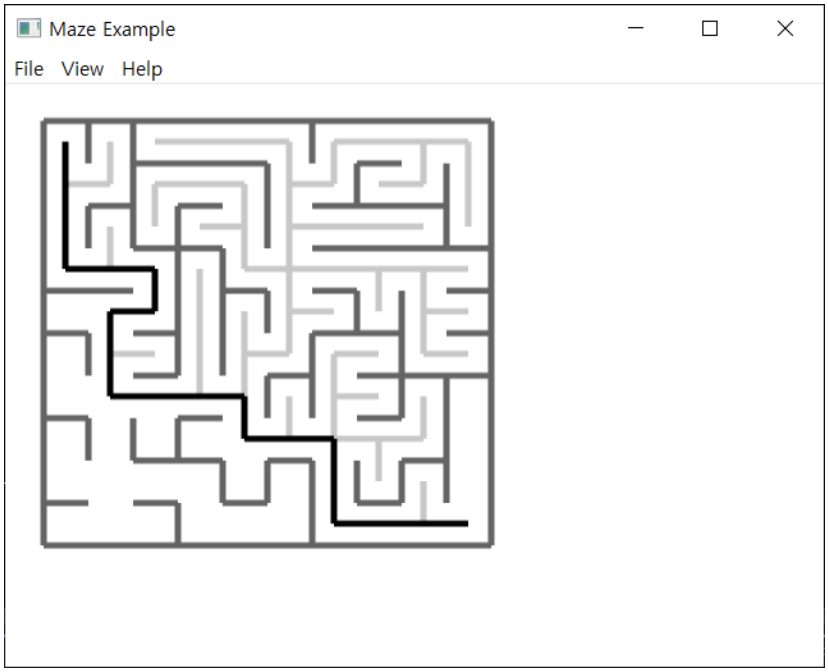
한 방향으로 계속 나아가다가 더 이상 전진할 수 없는 경우 이전 위치로 돌아와서 다시 갈 수 있는 방향을 찾아 이동한다. 이런 방식으로 도착점을 찾을 때까지 반복한다. DFS를 구현한 자료구조가 인접 리스트라고 가정했을 때, v개의 vertex와 e개의 edge를 갖는다고 가정한다면, DFS의 시간 복잡도는 O(v + e)이다. 왜냐하면 worst case의 경우 모든 vertex와 edge를 방문하게 되기 때문이다.

BFS는 Breath First Search의 줄임말로 너비 우선 탐색 방식이다. BFS는 시작점에서 출발하여 동일한 거리의 지점들을 방문하는 방식으로 도착점을 찾는 방식이다. 이런 방식의 특성상 최단 거리를 구하기 위한 목적에서 DFS보다 효율적이다. 그러나 BFS 역시 worst case의 경우 모든 vertex와 edge를 방문하게 된다. 따라서 BFS의 시간 복잡도 역시 O(v+e)로 나타낼 수 있다.

우선 구조체 node가 vertex의 역할을 하며 구조체 node에 포함된 node_pointer link가 edge에 해당한다. 이 때 link를 4개의 1차원 배열로 정의했는데 4개는 4개의 방향을 각각 의미한다. row와 col은 파일에서 정보를 읽어와서 저장했던 이차원 배열에서 값을 비교하고 저장할 때 사용하기 위해 활용되며 index는 좌측 상단부터 해당 vertex가 몇 번째인지를 나타내준다. Middle_x와 middle_y는 bfs와 dfs를 구한 이후 윈도우에 그릴 때 사용하기 활용된다.

DFS는 stack을 이용한다. 또한 해당 vertex를 방문했는지를 표시하기 위해 visited를 정의하고 활용한다. 현재의 위치에서 위쪽, 오른쪽, 아래쪽, 왼쪽 순으로 이동할 수 있는지 탐색하고 이동할 수 있다면 해당 노드를 스택에 push하고 동일한 시행을 반복한다. 그러다가 만약 이동하지 못하는 경우에는 이전 위치로 돌아가기 위해 스택에서 pop을 하여 이전 위치를 불러오고 이전 위치에서 다시 4방향으로 이동할 수 있는지 탐색하여 이전에 이동하지 않았던 방향으로 이동한다. 이런 방식으로 계속 반복하다가 도착점을 찾는 경우 while loop을 나오고 DFS 경로를 찾았다는 flag를 업데이트 한다.

BFS도 DFS와 유사하지만 stack말고 queue를 이용한다. 시작점을 queue에 add하고 시작하며 front가 rear와 같아질 때까지, 즉 queue가 비워질 때까지 while loop을 돌게 된다. While 안에서는 queue의 값을 하나 delete하여 해당 위치로 삼고 해당 위치에서 4가지 방향으로 이동할 수 있는 경우마다 queue에 더한다. 이동할 수 있는지 판단하는 것에는 edge에 해당하는 link가 NULL이 아닌지와 이동하려는 노드의 visited가 0인지가 고려된다. 이런 방식으로 queue에 노드를 더하고 빼면서 탈출구를 찾을 때까지 반복하게 된다.



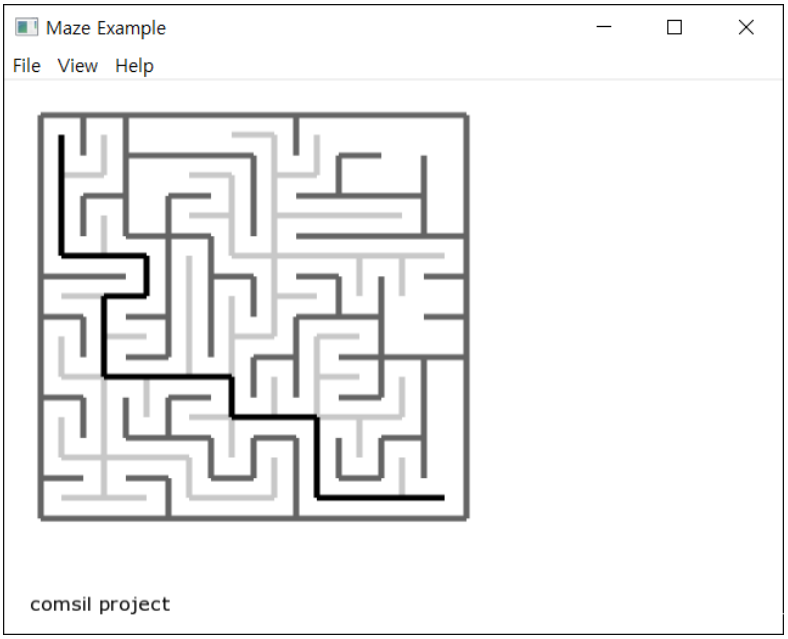
<DFS로 구현한 경로>

실질적인 DFS로 경로를 구하기 이전에 필요한 작업들이다. Visited는 노드의 방문 여부를 저장하고 stack은 시작점부터 도착점에 도달할 때까지의 경로를 저장하는 용도로, all은 stack에 남지 않는 모든 경로를 남겨 놓기 위한 용도로 활용된다. Visited를 0으로 초기화하여 방문하지 않았음을 의미하게 하고 0번째 노드, 즉 좌측 상단의 노드를 시작점으로 하기에 visited[0]에 1을 대입하고 해당 위치를 스택에 push한다.

DFS를 구하는 실질적인 부분은 두 개의 while loop으로 이루어져 있다. Outer loop은 도착점을 찾을 때 나오게 설정하고 inner loop은 4개의 방향을 모두 탐색한 경우 나오게 설정한다. 스택에 저장되어 있는 위치를 pop해서 불러오고 해당 위치를 ptr로 가리킨다. 그리고 해당 노드의 4가지 link를 탐색하게 되는데, 이 때는 3가지 경우가 존재한다. 첫 번째는 벽이라서 혹은 이미 지나온 곳이라서 가지 못하는 경우이다. 이 때는 다음 방향을 탐색하면 되므로 dir에 1을 더하기만 한다. 두 번째 경우는 탈출구에 도착한 경우로 해당 노드로 이동한 다음 마지막 위치를 push하고 found를 true로 변경하여 outer loop을 벗어날 수 있도록 한다. 마지막 경우는 첫 번째도 두 번째도 아닌 경우로 이동할 수 있는 경우를 의미한다. 이 때는 해당 노드로 이동하고 위치를 스택에 push하고 visited도 1로 변경하고 dir은 0으로 변경시킨다. Dir을 0으로 변경시키는 이유는 새로운 노드로 넘어갔으니 4 방향을 처음부터 조사할 수 있도록 하기 위함이다.

DFS의 시간 복잡도는 노드의 개수를 v, 노드끼리 연결하는 edge를 e라고 할 때 O(v+e)이다. 왜냐하면 worst case의 경우 모든 노드와 연결 링크를 탐색하게 되기 때문이다. 공간 복잡도는 노드의

개수가 n개일 때 O(n)이다. 내가 선언한 자료구조에 따르면, 노드 안에 4가지 방향에 해당하는 값도 포함되어 있다. 따라서 n개의 노드가 있다고 한다면 O(n)의 공간 복잡도를 갖는다.



<BFS로 구현한 경로>

BFS도 DFS와 유사한데 차이점은 queue를 활용한다는 것이다. Stack도 동적 할당한 것을 확인할 수 있는데 이것은 BFS로 경로를 탐색한 이후 최단 경로가 어떻게 되는지 이후에 다시 구하는 과정에서 사용하기 위해 사용했다. DFS와 유사하게 첫 번째 노드를 queue에 더하고 시작한다.

BFS는 while loop을 outer loop으로 for loop을 inner loop으로 갖는 방식으로 구현했다. Outer loop은 front가 rear와 같은 경우 나오게 되는데, 이 경우는 queue에 있는 값들을 모두 탐색한 경우이다. While loop에 들어가서 queue에 있는 값을 하나 지우고 해당 위치를 받아와 ptr로 가리킨다. 그리고 inner loop인 for loop에서 4가지 방향으로 조사하여 벽도 아니고 방문한 곳도 아닌 곳, 즉 이동할 수 있는 방향의 값들을 모두 queue에 add한다. 이렇게 queue에 노드들을 더하다가 탈출구에 도달한 경우 found를 true로 바꾼다. 이 때 중요한 것은 found가 true가 되었더라도 break를 따로 사용하지 않아 탈출구와 동일한 거리의 다른 지점들도 queue에 더할 수 있도록 한다.

BFS의 시간 복잡도는 DFS와 동일하게 노드의 개수를 v, 노드끼리 연결하는 edge를 e라고 할 때 O(v+e)이다. 왜냐하면 worst case의 경우 모든 노드와 연결 링크를 탐색하게 되기 때문이다. 공간 복잡도는 노드의 개수가 n개일 때 O(n)이다. 내가 선언한 자료구조에 따르면, 노드 안에 4가지 방향에 해당하는 값도 포함되어 있다. 따라서 n개의 노드가 있다고 한다면 O(n)의 공간 복잡도를

갖는다.

4. 추가/변경 사항

4.1 목록 정리

(1) make random maze 세부 메뉴 만들기

(2) make_maze( )함수를 통해 HEIGHT와 WIDTH를 입력 받아서 새로운 미로 파일을 생성

(3) 종착점에는 치즈를, 시작점에는 쥐구멍의 모습을 그려서 표시

(4) 미로의 시작점과 끝나는 점을 랜덤하게 배치

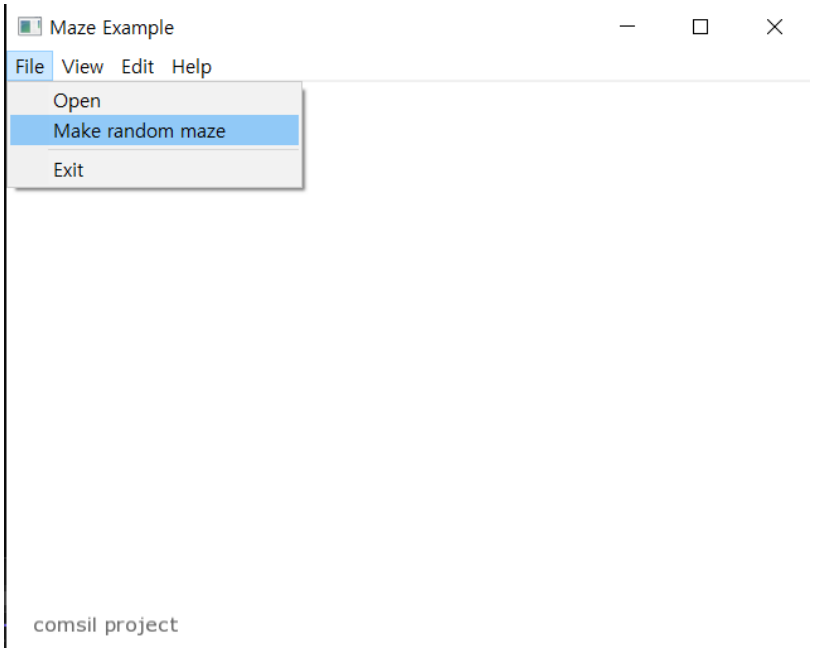
(5) DFS, BFS 이후에 Clear 기능 추가(미로만 남기기)

(6) DFS, BFS 애니메이션 생성

4.2 목록별 상세 정보

(1) make random maze 세부 메뉴 만들기

미로 프로그램 윈도우창에 메뉴와 버튼을 추가하기 위해서는 HMENU라는 클래스에 포함된 CreateMenu함수를 사용해야 한다. CreateMenu함수를 사용하여 함수가 성공하면 새로 만든 메뉴 핸들을 반환하고 함수가 실패하면 NULL을 반환하는 함수이다. 처음 만든 상황에서는 비워져 있는 상황인데 InsertMenuItem, AppendMenu, InsertMenu함수를 사용하여 원하는 방식으로 설계할 수 있다. 그리고 CreateMenu와 구분되는 CreatePopupMenu함수도 사용한다. CreateMenu가 전체 메뉴창을 만든다면 CreatePopupMenu함수는 역할은 비슷하지만 메뉴창 안의 세부 메뉴들을 만드는데 사용된다. 또한 세부 메뉴를 만들 때 AppendMenu와 InsertMenu함수를 사용하여 메뉴 항목을 삽입하고 추가한다. 이를 바탕으로 윈도우 창 상단에 make 메뉴와 세부 메뉴인 make random maze를 만들고 해당 메뉴를 클릭했을 때 원하는 작업을 수행하기 위해서 다음과 같은 구현이 필요하다. menu->AddPopupItem(hPopup, "Make random maze", false, false);



<Make Random maze 세부 메뉴 구현>

(2) make_maze( )함수를 통해 HEIGHT와 WIDTH를 입력 받아서 새로운 미로 파일을 생성

void make_maze()

temp_height와 temp_width라는 지역변수를 사용하여 입력 받은 너비와 높이를 저장한다. 이를 토대로 이차원 배열 temp_maze에 temp_width x temp_height 크기의 메모리를 동적 할당하고 차례대로 Init_maze, Eller_algorithm, print_maze, fwrite_maze를 실행하여 미로를 maze.maz 파일에 저장한다. 마지막으로 동적 할당했던 temp_maze를 해제하고 종료한다. 시간 복잡도는 make_maze가 포함한 Eller_algorithm의 시간 복잡도에 영향으로 n이 미로 방의 개수라고 한다면, O(n^2)이 된다. 공간 복잡도는 이차원 배열인 temp_maze를 사용하므로 O(n)이 된다.

void Init_maze(int temp_height, int temp_width, int** maze)

make_maze함수에서 호출되어 사용되는 함수로 maze를 초기화하는 역할을 한다. 행이 짝수, 열도 짝수인 경우에는 모서리를 의미하는 1을 대입하고, 행이 짝수, 열이 홀수인 경우에는 가로벽을 의미하는 2를 대입하고, 행이 홀수, 열이 짝수인 경우에는 세로벽을 의미하는 3을 대입하고, 행이 홀수, 열이 홀수이 경우에는 방을 의미하는 0을 대입한다.

void print_maze(int temp_height, int temp_width, int** maze)

지역 변수 command를 사용하는데 command는 maze의 값, 즉 0부터 4사이의 값을 받아서 switch 문을 이용하는데 사용된다. Init_maze에서 설명했던 설정대로 1은 모서리, 2는 가로벽, 3은 세로벽으로 간주한다. 추가된 점은 Eller Algorithm에서 설명되지만 4는 가로벽이나 세로벽 중 허물어져서 빈칸인 경우를 의미하며 이 경우에는 빈칸을 출력한다. 그 외의 경우는 미로의 방으로 간주하여 역시 빈칸으로 출력한다. 이 함수의 목적은 윈도우에 출력하는 것이 아니라 콘솔에 미로의 생김새를 출력하여 디버깅이나 이용의 편이를 위해 활용한 것이다.

void fwrite_maze(int temp_height, int temp_width, int** maze)

위의 print_maze와 완전히 동일한 원리로 작동하는 함수인데, ofstream writeFile을 사용하여 파일을 생성하고 정보를 저장하는 역할을 하는 함수이다. 상대 경로를 사용하여 data 파일 안에 maze.maz 파일을 생성하고 미로의 정보를 저장한 다음 성공적으로 작동했으면 Making Radom Maze Succeed를 출력하고 종료한다.

void Eller_algorithm(int temp_height, int temp_width, int** maze)

int select: 랜덤으로 0, 1을 설정해서 벽을 지울지 말지 결정할 때 사용하는 변수

int number = 10: 미로의 방의 집합을 구분하기 위해 사용하는 변수(10부터 시작해서 집합이 새로 등장할 때마다 1씩 증가해서 설정해준다)

int flag = 0: 행에서 같은 집합에 속한 방이 수직 경로를 내렸는지 판단하는 변수

int possible = 0: 다음 단계에서 수직 경로를 내릴 수 있는지 판단하는 변수

int temp: 두 인접한 미로의 방이 벽을 허물고 집합을 합칠 때 임시로 값을 저장하기 위해 사용하는 변수

첫 번째 줄에서 서로 다른 집합에 속하도록 초기화를 하고 첫 번째 줄을 대상으로 벽을 랜덤하게 제거하거나 유지하도록 한다. 세로벽이 지워진 부분은 4로 간주하기 때문에 4를 대입하고 벽이 없어지면 벽 기준 왼쪽으로 같은 집합에 속하도록 maze의 값을 대입해 준다.

그 다음으로 마지막 줄 전까지 반복적으로 수행하는 for문이 2개 중첩되어 들어오는데 outer loop은 마지막 줄 전까지 반복하는 것을 설정하고 inner loop의 첫 번째는 수직 경로를 만들기 위한 것, inner loop의 두 번째는 수직 경로가 내려진 후 아래 줄을 읽어서 수직 경로가 없던 곳에서 새로운 집합을 만들고 해당 행에서 임의로 벽을 제거하는 작업을 수행한다.

마지막 줄의 경우에는 인접해 있으면서 서로 다른 집합에 속한 미로의 방들 사이의 모든 벽을 제거하고 앞과 동일하게 벽이 허물어지면 왼쪽을 기준으로 오른쪽 방의 집합을 변경해주고 동일하게 오른쪽 방과 같은 집합에 속해 있던 방들의 집합도 왼쪽 방의 집합에 통합될 수 있도록 값을

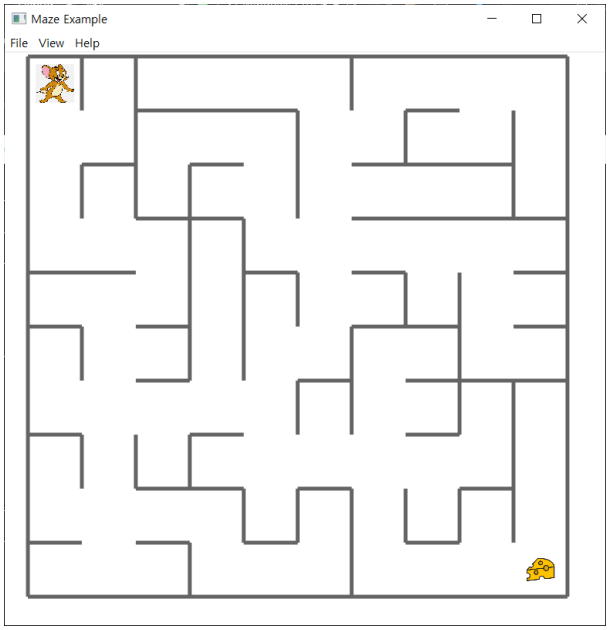
변경한다.

(3) 시작점에는 제리를, 도착점에는 치즈의 이미지 그리기

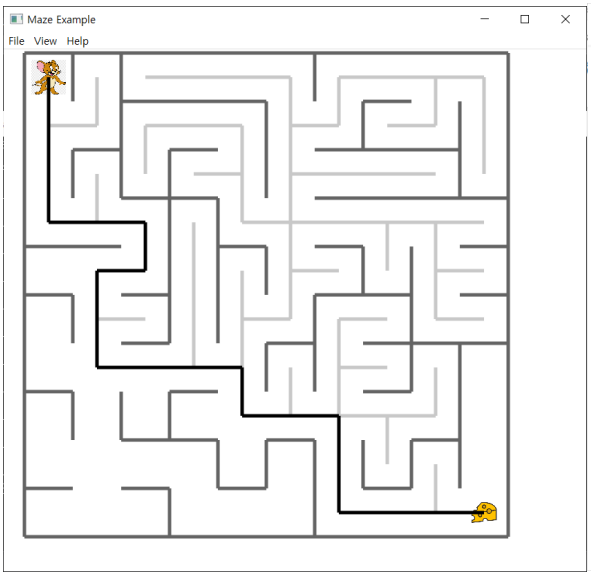
시작점과 도착점을 랜덤하게 설정하는 기능도 추가할 예정이기 때문에 시작점과 도착점을 구분해서 보여줄 장치가 필요하다. 그래서 시작점에는 쥐를 표현하고자 '톰과 제리'의 제리의 이미지를, 도착점에는 제리가 좋아하는 치즈의 이미지를 그리고자 한다. 이를 위해서는 draw 함수를 수정해야 한다.

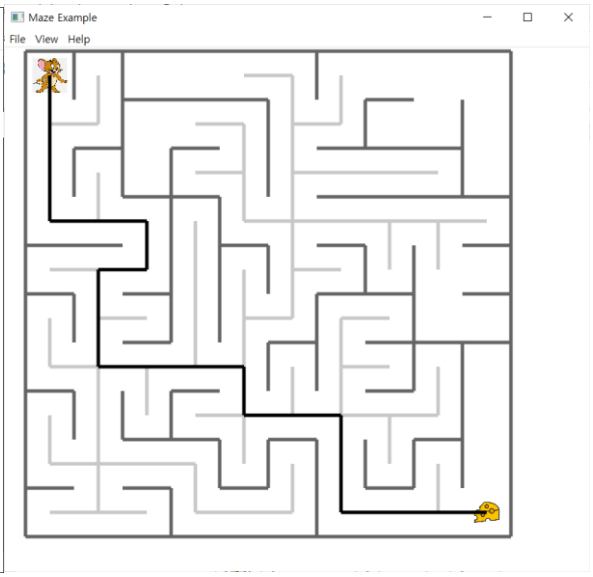
기존의 draw함수는 미로의 정보에 따라 미로의 형태를 완성한 이후, isDFS와 isBFS의 값에 따라 dfsdraw나 bfsdraw함수를 호출하여 화면에 경로 탐색의 결과물을 그렸다. 시작점과 도착점을 나타내는 기능은 isDFS나 isBFS와는 별개이며 파일이 열려서 미로의 정보가 확정되는 것에 의존하기 때문에 isOpen의 값에 따라 만약 isOpen이 1이면 파일이 열렸다고 판단하여 윈도우에 미로를 그리는 기능 다음에 구현하면 된다.

Dfsdraw나 bfsdraw에서 사용하기 위해 이미 미로의 방(노드)의 중심의 좌표에 해당하는 middle_x와 middle_y를 저장하였다. 따라서 이를 활용하면 되는데 한 가지 문제점은 Image.draw로 이미지를 윈도우에 출력할 때 input은 기준이 되는 점의 x, y좌표와 이미지의 가로, 세로의 길이로 총 4개이다. 그런데 기준이 되는 점의 x, y좌표는 이미지의 좌측 상단으로 고정되어 있기 때문에 기존의 middle_x와 middle_y를 사용하면 원하는 지점에 제리와 치즈의 이미지를 위치시킬 수 없다. 따라서 기준이 되는 점의 x, y좌표를 이미지의 좌측 상단에서 이미지의 정중앙으로 바꾸어주는 작업이 필요하며 이는 ofSetRecMode(OF_RECTMODE_CENTER)을 통해 구현 가능하다. 이러한 원리에 따라 파일을 열었을 때, DFS나 BFS를 실행했을 때의 결과는 아래와 같다.



<파일을 열었을 때>





<DFS 실행했을 때>

<BFS 실행했을 때>

(4) 미로의 시작점과 도착점을 랜덤하게 배치

기존의 미로는 좌측 상단을 출발점 혹은 시작점으로 가정하고 미로의 우측 하단을 도착점 혹은 탈출구로 가정하고 있다. 이러한 가정이 DFS와 BFS 함수의 작동 원리를 보여주기에 용이했으며 경로를 탐색할 때 미로의 상당 부분을 탐색하면서 DFS와 BFS가 제대로 작동하는지 파악하기에도 용이했다. 하지만 이번 프로젝트에서 구현한 미로는 완전 미로로 어떤 임의의 두 지점을 선택하더라도 경로가 있어야만 하는 특수한 미로의 성질을 가지고 있다. 이러한 미로의 성질을 잘 보여

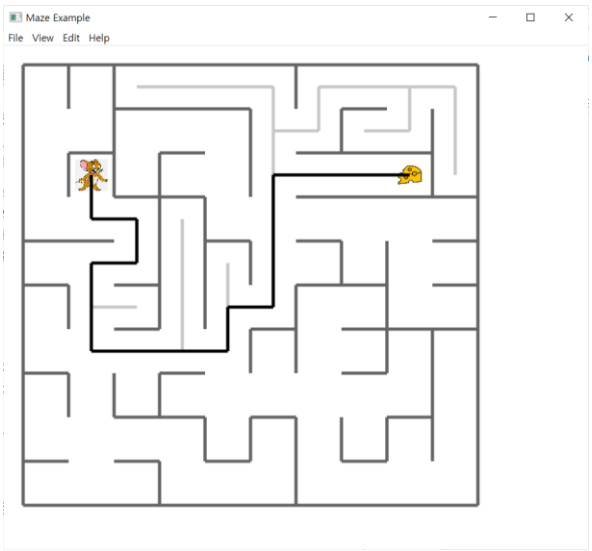
주기 위해서는 시작점과 도착점이 단순히 좌측 상단과 우측 하단으로 고정되어 있으면 안 된다. 따라서 최종 프로젝트를 위한 개선 사항의 일환으로 완전 미로의 특성을 더 잘 보여줄 수 있도록 미로의 시작점과 도착점을 랜덤하게 설정할 수 있는 기능을 추가하였다.

이를 위해 멤버 변수에 entrance와 exit을 추가했다. Entrance는 시작점을, exit은 도착점을 의미한다. 처음에는 setup함수에서 entrance를 0으로 exit을 num_of_terms -1로 설정하려 했지만 이는 setup함수가 프로그램의 실행과 동시에 결정되는 것이기 때문에 아직 num_of_terms가 정해지지 않아 오류를 만들어냈다. 오류를 없애고 의도대로 작동시키기 위해서는 num_of_terms가 정해지는 readFile 내부에서 entrance를 0으로, exit을 num_of_terms -1로 초기화해야 한다.

다음으로 (1)에서 세부 메뉴를 만드는 것과 동일한 방식으로 Edit 메뉴를 만들고 세부 메뉴로 Random ENTRANCE, Random EXIT을 만든다. appMenuFunction에서는 Random ENTRANCE가 선택될 때 entrance에 rand( ) % num_of_terms를 대입한다. 같은 맥락에서 Random EXIT이 선택될 때는 exit에 rand( ) % num_of_terms를 대입한다. 이를 토대로 새로운 entrance와 exit을 설정할 수 있다.

추가적으로 수정해야 할 것은 DFS와 BFS 함수이다. 기존의 DFS와 BFS는 시작점을 0번째 index의 노드로, 도착점을 num_of_terms – 1번째 index의 노드로 가정을 하였다. 따라서 해당하는 부분들을 entrance와 exit으로 대체해야 DFS와 BFS 함수가 정해진 의도에 맞게 작동할 수 있게 된다.

나아가 시작점을 나타내는 Jerry의 이미지와 도착점을 나타내는 Cheese의 이미지의 위치를 그리는 함수에서도 DFS, BFS 함수에서 수정한 것과 동일하게 0을 나타내던 값은 entrance로 num_of_terms -1을 나타내던 값은 exit으로 대체하여 시작점과 도착점이 변할 때, 이에 맞추어 이미지도 해당 노드로 이동할 수 있도록 한다.



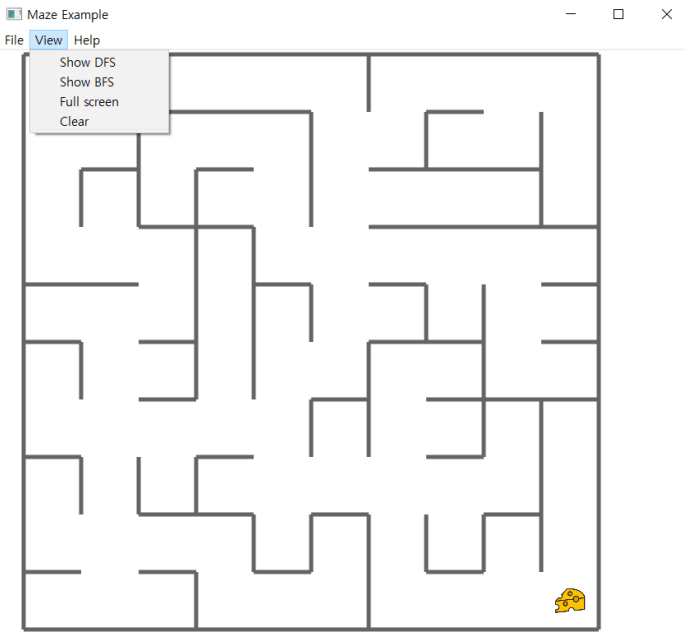
<시작점 도착점 변경 후 DFS>

<시작점 도착점 변경후 BFS>

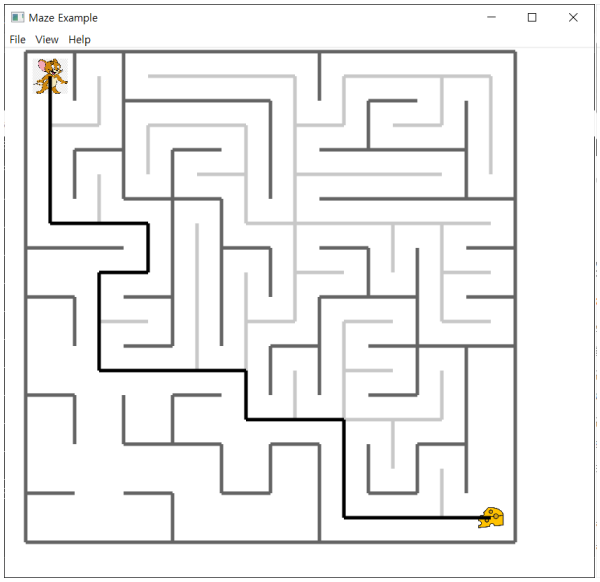
(5) DFS, BFS 이후에 Clear 기능 추가(미로만 남기기)

미로 프로그램에서 DFS와 BFS 두 가지 기능으로 경로를 표현할 수 있으며 한 번 경로를 그리면 미로 위에 경로가 그려지기 때문에 이를 지우기 위해서 프로그램을 껐다가 다시 켜야 하는 상황이 벌어진다. 이를 해결하기 위해서 DFS와 BFS가 화면에 그려졌을 때 화면에 그려진 경로를 없애는 Clear 기능을 추가하여 프로그램을 재시작하지 않아도 DFS와 BFS를 껐다가 키거나 다른 기능을 실행하는데 용이하도록 프로그램을 개선하였다.

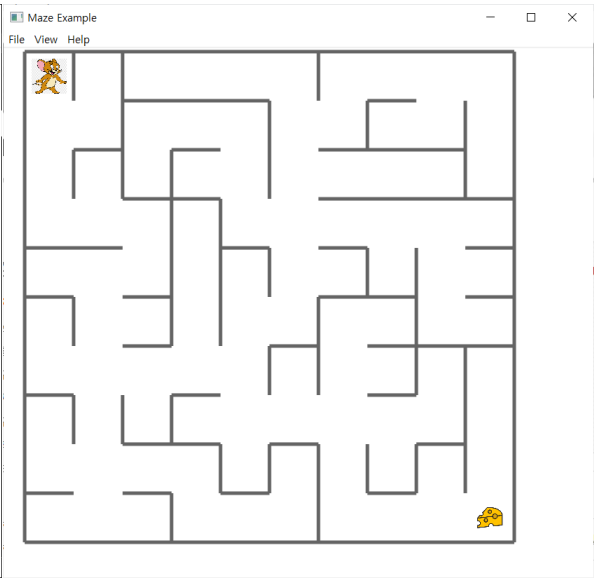
이를 구현하기 위해서는 setup함수와 appMenuFunction을 조작하면 된다. 우선 setup함수에서 View 메뉴의 세부 메뉴로 Clear 메뉴를 추가한다. 그리고 appMenuFunction에서 clear 메뉴가 선택되었을 때 isDFS와 isBFS를 모두 0으로 초기화하면 된다. 이처럼 간단한 조작으로 화면 clear 기능이 추가되는 것은 draw 함수에서 DFS와 BFS가 어떻게 작동하는지를 알면 쉽게 이해할 수 있다. draw함수에서는 isDFS와 isBFS가 DFS와 BFS가 성공적으로 완료되었다는 flag 역할을 하며 isDFS와 isBFS가 1일 때 dfsdraw와 bfsdraw를 호출하도록 설정되어 있다. 따라서 flag 역할의 isDFS와 isBFS만 0으로 초기화한다면 dfsdraw와 bfsdraw가 호출되지 않아서 매 프레임별로 윈도우 화면에 그려지던 경로를 없앨 수 있는 것이다.



<Clear 메뉴를 추가>



<DFS 실행 화면>



<Clear한 이후 화면>

(6) DFS, BFS 애니메이션 생성

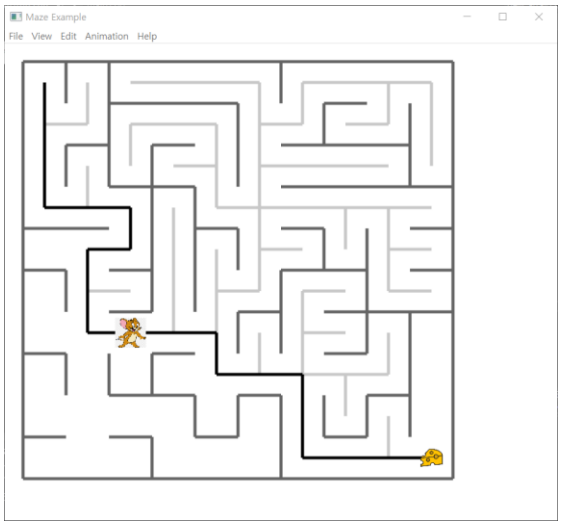
DFS와 BFS 경로가 구해지면 이것을 따라 출발점에 있던 제리가 움직이면서 치즈가 있는 도착점에 가는 애니메이션을 구현하고자 했다. 이를 구현하기 위해서 가장 핵심이 되는 것은 update 함수를 사용하는 것이다.

그 전에 DFS와 BFS Animation을 실행시키기 위한 메뉴를 만들고 각각의 메뉴가 눌렸을 때에는 isDFS_Animation과 isBFS_Animation을 1로 설정하고 current_top을 0으로 설정하여 원하는 경로 탐색의 애니메이션을 만들기 위한 준비를 마친다.

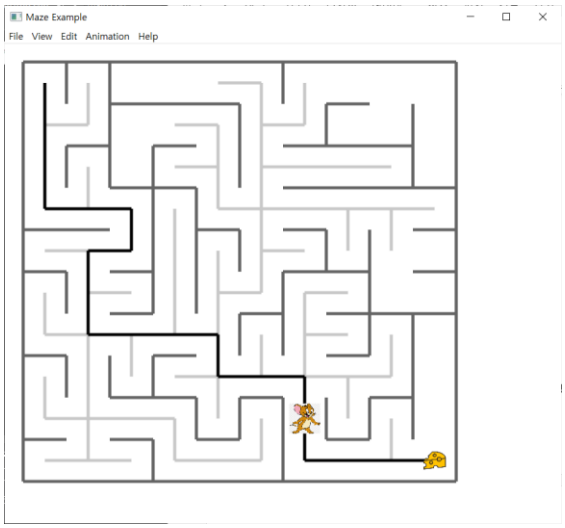
update함수에서는 1로 설정된 flag, 즉 isDFS_Animation이나 isBFS_Animation이 활성화되었을 때 stack의 값을 current로 받아와서 시간이 지남에 따라 current의 값이 stack의 값이 1개씩 바뀜에 따라 따라서 바뀌도록 설정해야 한다. 이 때 중요한 것은 시간이 지남에 따라서 변하는데 시간이 지날 때 너무 빨리 변해버린다면 애니메이션의 결과를 확인하기 힘들 것이다. 따라서 ofSetFrameRate(3)을 설정하여 해당 애니메이션 기능이 활성화되는 동안에는 애니메이션이 천천히 작동하여 확실히 인지할 수 있도록 한다. 그리고 애니메이션이 끝나는 것은 제리가 치즈가 있는 도착점에 도착했을 때인데, 이 때는 isDFS_Animation이나 isBFS_Animation을 0으로 재설정하여 애니메이션이 자동으로 멈추도록 설정하고 ofSetFrameRate(60)으로 설정하여 다른 작업들을 할 때 프레임 속도가 늦어짐에 따라 발생할 수 있는 문제들이 생기지 않도록 해야한다.

마지막으로 수정해야 할 부분은 기존의 Jerry.draw와 Cheese.draw가 있는 draw함수이다. 기존에는 entrance와 exit에 고정되어 있기만 했으면 되니깐 entrance와 exit의 위치에 맞게 그리도록 했는

데 이제는 만약 애니메이션이 활성화된다면 치즈는 그대로 있더라도 제리는 경로를 따라 움직여야 하기 때문에 이 경우에만 Jerry.draw의 입력으로 current에 해당하는 노드의 위치를 넣을 수 있도록 변경해야 한다.



<DFS Animation 결과>



<BFS Animation 결과>

5. 느낀 점 및 개선 사항

5.1 느낀 점

이번 미로 프로젝트를 통해 습득한 점들이 있다. 우선 가장 신선하고 새로웠던 것은 윈도우 창을 만들고 윈도우 창에 메뉴와 세부 메뉴, 버튼 등을 만드는 것을 배운 것이다. 실습에서 실질적으로 윈도우와 메뉴를 만드는 부분을 수정하지는 않았지만 기존에 단순히 화면에 결과를 띄우거나 특정 키를 입력 받아, 혹은 마우스 클릭으로 결과를 출력하게끔 하는 것과 달리 윈도우 창에서 메뉴 창을 선택하는 방식을 구현한 것이 새로웠고 해당 부분을 읽고 이해하는 과정에서 배워야 할 부분이 많다는 생각이 들었다.

두 번째로 배운 점은 미로의 자료 구조를 설계할 때 이후에 추가될 기능을 고려하여 graph로 구현하는 과정과 관련되어 있다. 미로 1주차 실습에서는 단순히 int형의 2차원 배열을 활용하여 미로를 만드는 프로그램을 설계했다. 그러나 이번 2주차 실습에서는 3주차에서 수행할 BFS와 DFS라는 알고리즘을 수행하기 위해 적절한 자료구조를 설계하는 것이 관건이었다. 때문에 단순히 2차원 배열을 동일하게 사용하는 것이 아니라 BFS와 DFS를 학습하고 어떤 원리로 작동하는 알고리즘인지 알아보고 그러한 작업을 원활하게, 효율적으로 수행하기 위한 자료 구조를 구상하고 그에 맞게 draw함수를 설계하는 과정을 거쳤다. 이러한 과정에서 단순히 프로그래밍을 할 때, 직관적으로 떠오르고 간단하고 쉬운 자료 구조를 사용하는 것이 아니라 최종적으로 어떤 역할을 수행

할지를 생각하고 그에 맞추어 자료구조부터 효율적인 것을 구상하는 과정이 이후 프로그래밍 학습에 큰 도움이 될 경험이었다고 생각한다.

이후에는 최종 프로젝트를 준비하면서 기존에 학습했던 OpenFramework의 기능들, 예를 들어 update함수나 draw함수를 잘 활용하여 미로의 애니메이션을 만드는 등 OpenFramework를 활용하는 방법에 대해 배웠다. 처음 OpenFramework를 접하였는데 처음에는 어색하고 이해되지 않고 어려웠지만 이번 미로 프로젝트를 준비하면서 이것 저것 많이 시도해보고 검색해보면서 OpenFramework를 활용하여 새로운 프로그램과 기능들을 만들 수 있겠다는 생각이 들었다.

5.2 개선 사항

공간 복잡도는 maze라는 이차원 배열을 설정하는 것과 이를 토대로 graph에 노드를 통한 adjacency list의 형태로 미로를 구현한 것이 주로 고려된다. 프로그램에서 설계했던 방식에 맞추어 본다면 미로의 너비를 w, 미로의 높이를 h로 했을 때 공간 복잡도는 O(h x w)가 될 것이고 graph에서 구현한 형태로 보아도 노드의 수를 n이라고 했을 때 O(n)이 된다. 그러나 h x w가 곧 노드의 개수인 n이 되기 때문에 전체 프로그램의 공간 복잡도는 O(n)이 된다. 물론 스택이나 큐를 이용하기 위해 동적 할당을 하지만 이것들의 공간 복잡도는 총 노드의 수가 n이라고 한다면 O(n)이기 때문에 추가적으로 공간 복잡도를 어렵게 만들지 않는다.

공간 복잡도는 기존에 생각했던 것에 맞추어 잘 설계되었지만 시간 복잡도에서 아쉬운 점이 있다. Adjacency list의 형태로 graph에 미로를 변환하여 저장하였기에 BFS나 DFS로 경로를 탐색할 때 시간 복잡도는 잘 구현했다고 생각하는데 readFile에서 엘러의 알고리즘을 활용하여 이차원 배열인 maze에 미로의 정보를 저장할 때 시간 복잡도가 O(n^2)가 되었다. 두 인접한 노드가 벽을 허물고 하나의 집합으로 통합되는 과정에서 다른 방법을 사용하여 효과적으로 처리한다면 보다 낮은 시간 복잡도로 구현할 수 있을 것으로 예상되는데, 이 부분이 잘 해결되지 못하였고 예상보다 시간 복잡도가 높아지는 문제가 발생했다. 이번 프로젝트 이후에 이 부분에 대해 탐구하고 개선할 수 있는 방법을 생각하는 것이 숙제가 될 것이라 생각한다.