컴퓨터공학설계및실험I(CSE3013-03)

20200368 김유신

PROJECT: BREAK THE MAZE

보고서 목차

1. 프로젝트 목표 및 실험 환경
2. 정의한 변수 및 함수
3. 프로젝트 전체 플로우차트
   + 자료구조/알고리즘 + 시간/공간 복잡도
4. 창의적 구현 항목
5. 프로젝트 실행 결과
6. 느낀 점 및 개선 가능한 사항

1. 프로젝트 목표 및 실험 환경

저는 어릴 적부터 게임을 즐겨 했고 자연스레 컴퓨터와 가까운 삶을 살게 되었습니다. 컴퓨터가 좋아져서 대학교에 와서도 컴퓨터공학을 복수 전공했지만, 수업에선 백날 지루한 코드 분석, 비트 계산 같은 것만 하고 제가 알고 싶었던 내용을 배우기가 어려웠습니다.

그러던 중 본 과목에서 Openframework 를 배운 순간 컴퓨터공학을 전공해 온 지난 2 년 반 동안의 시간 중 가장 신선한 충격을 받았습니다. 그래픽을 어떻게 다루는지 몸소 체험할 수 있었고, 제가 즐겨 하는 게임들이 결국은 다 이런 기초적인 기법들을 기반으로 만들어졌겠구나 싶어 정말 큰 흥미를 느끼게 되었습니다. 그래서 저는 단순하지만 중독성 있는 게임을 이번 프로젝트에서 구현해 보기로 결심하였습니다. 마지막 3 주 동안 진행한 미로 프로젝트에서 완전 미로 구현 및 미로를 그래픽으로 나타내는 방법을 학습하였으니, 이젠 그 미로를 하나씩 부숴 보겠습니다.

제가 프로젝트를 통해 만들어낸 것은 미로를 부수는 Break the Maze 라는 게임입니다. 이 게임의 주인공인 광부는 미로를 탐험하면서 에너지 드링크를 찾아내 얻은 파워로 미로의 안에 있는 벽을 하나씩 부숴 나갑니다. 모든 벽이 부숴진다면 게임은 종료됩니다.

프로젝트를 진행한 실험 환경은 다음과 같습니다.

1. Hardware:

   Intel® Core™ i7-1065G7 CPU @ 1.30GHz 1.50 GHz

   24.0 GB (RAM)

   Intel Iris Plus Graphics (GPU)

2. OS: Windows 11 Home 22621.1702
3. IDE: Visual Studio 2019, Win32
4. Software: Openframework

2. 정의한 변수 및 함수

지금부터는 게임 구현을 위해 정의한 변수 및 함수에 대해 설명해 보겠습니다.

```
13  #define MAIN        0
14  #define EASY_MODE   1
15  #define HARD_MODE   2
16  #define RESULT      3
17
18  #define UP      0
19  #define DOWN    1
20  #define LEFT    2
21  #define RIGHT   3
22
23  #define LEFT_X      212     // 미로의 좌상단 꼭짓점의 X 좌표
24  #define TOP_Y       84      // 미로의 좌상단 꼭짓점의 Y 좌표
25  #define RIGHT_X     812     // 미로의 우하단 꼭짓점의 X 좌표
26  #define BOTTOM_Y    684     // 미로의 우하단 꼭짓점의 Y 좌표
27  #define MARGIN      4       // Anti-aliasing을 위한 margin
28
29  #define MAZE_LENGTH (RIGHT_X - LEFT_X)  // 게임 플레이 화면에 그려지는 미로의 너비(= 높이)
```

먼저 기본적인 매크로입니다.

1. MAIN, EASY_MODE, HARD_MODE, RESULT: 각각 ofApp::draw()에서 메인 화면, 이지 모드 화면, 하드 모드 화면, 게임 결과 화면을 그리기 위한 flag 변수에 저장하지 위한 값입니다.
2. UP, DOWN, LEFT, RIGHT: 캐릭터가 움직이는 방향, 향하려는 방향 등을 나타내는 값입니다.
3. LEFT_X, TOP_Y, RIGHT_X, BOTTOM_Y: 네모낳게 그려지는 미로의 가장 바깥 윤곽선이 그리는 직사각형을 기준으로, LEFT_X 는 직사각형 맨 왼쪽의 X 값이고, RIGHT_X 는 맨 오른쪽의 X 값입니다. TOP_Y 는 가장 위쪽의 Y 값이고, BOTTOM_Y 는 가장 아래쪽의 Y 값이 됩니다.
4. MARGIN: 저는 미로를 그리기 위해 각각의 위치에서 선을 그렸는데, 선의 굵기가 있기 때문에 모서리가 계단처럼 울퉁불퉁하게 보이게 됩니다. 이를 컴퓨터 그래픽에서 계단 현상(Aliasing)이라고 하는데, 이를 제거하기 위해 모든 선을 양쪽으로 일정 pixel 더 길게 그렸습니다. 이렇게 추가적으로 넓힌 간격이 MARGIN 입니다. 이 값은 이 외에도 그릴 이미지의 크기를 결정하거나 캐릭터가 이동하고자 하는 방향을 나타내는 화살표의 크기를 결정할 때 임의의 값으로 쓰였습니다.
5. MAZE_LENGTH: 미로의 너비이자 높이입니다. (미로는 정사각형이므로)

```
51          /* 미로를 다루는 함수들 */
52          void initializeMaze(int game_mode);     // 게임 모드에 따라 미로 및 관련된 변수들을 초기화하는 함수
53          void createMaze(int row, int col);      // recursive backtracking algorithm을 이용하여 완전 미로를 구성하는 함수
54          void deinitializeMaze();                // 미로와 관련된 데이터들을 원상태로 되돌리는 함수
55
56          void drawGameScreen();  // 게임 화면을 그리는 함수
57
58          /* 미로와 직접적으로 관련된 데이터들 */
59          vector< vector< char > > maze;  // 미로의 요소를 나타내는 2차원 벡터
60          vector< vector< bool > > visited;   // 완전 미로를 구성하기 위해 쓰이는 방문 여부 벡터
61          int HEIGHT;     // 미로의 높이(방의 개수만 고려)
62          int WIDTH;      // 미로의 너비(방의 개수만 고려)
63          int maze_row;   // maze 벡터의 행 개수(벽 포함)
64          int maze_col;   // maze 벡터의 열 개수(벽 포함)
65          unsigned int total_wall = 0;        // 미로에 존재하는 벽의 총 개수(0이 되면 게임 종료)
66
67          int step;   // 매우 중요한 값: 화면 상에서 미로가 그려지기 시작하는 (LEFT_X, TOP_Y)로부터 maze의 각 원소가 얼마나 떨어져 있는지 나타낸다.
68          int image_size;     // 그릴 entity의 이미지 너비 및 높이
69          /* 미로와 직접적으로 관련된 데이터들 */
```

6. maze: 미로의 기초적인 요소들(모서리, 벽, 방)을 character 로 저장하고 있는 2 차원 벡터입니다.
7. visited: 매 게임 시작 시마다 임의의 완전 미로를 구성하기 위해 미로 생성 알고리즘을 수행해야 하는데, recursive backtracking algorithm 으로 구현하였기 때문에 사용되는 visited 벡터입니다.
8. HEIGHT: 미로의 세로 방향으로 존재하는 방의 개수입니다.
9. WIDTH: 미로의 가로 방향으로 존재하는 방의 개수입니다.
10. maze_row: maze 벡터의 row 개수입니다.
11. maze_col: maze 벡터의 column 개수입니다.
12. total_wall: 미로에 존재하는 벽의 개수입니다. 참고로 이 미로 게임에선 윤곽선은 벽으로 고려하지 않고 윤곽선 안에 존재하는 선만을 벽으로 간주합니다.
13. step: 주석에도 나와 있듯이 매우 중요한 변수로, 미로의 좌상단 꼭짓점이 그려지는 위치로부터 가로 또는 세로로 k offset 만큼 떨어진 위치에 있는 변수가 화면 상에선 얼마나 떨어져 있는지를 계산하기 위해 쓰입니다. 예를 들어서 어떤 요소가 가로로 M, 세로로 N 만큼 떨어져 있고 미로의 좌상단 꼭짓점의 좌표가 (X, Y) = (0, 0)이라면 해당 요소의 위치는 (0 + step * M, 0 + step * N)입니다.
14. image_size: 미로 위에 그릴 캐릭터와 아이템 이미지(정사각형)의 한 변의 길이입니다.
15. initializeMaze(int game_mode): 매개변수로 주어지는 게임 모드에 따라 미로의 크기를 달리 하기 위하여 값을 설정하는 함수입니다. 이지 모드는 미로가 5 X 5 이고, 하드 모드는 10 X 10 입니다.
16. createMaze(int row, int col): recursive backtracking algorithm 으로 완전 미로를 구성하여 그 정보를 maze 에 저장하는 함수입니다.
17. deinitializeMaze(): 미로와 관련된 정보들을 게임 종료 시 초기화하여 예기치 않은 데이터 간 충돌을 방지하는 함수입니다.
18. drawGameScreen(): 미로와 캐릭터, 아이템을 그리는 함수입니다.

```
71          /* 게임 스테이터스를 다루는 함수들 */
72          void initializeGame();
73          void deinitializeGame();
74
75          /* 게임 플레이와 관련된 데이터들 */
76          unsigned int available_power = 0;   // 벽을 부술 수 있는 횟수
77          unsigned int total_energy = 0;      // 지금까지 획득한 에너지 개수: 플레이어 속도를 올려줌
78          int directing = -1; // 현재 누르고 있는 방향키 종류 저장
79
80          /* 캐릭터를 부드럽게 이동시키기 위한 변수 */
81          bool sliding = false;   // 캐릭터가 슬라이딩 상태인지 판단하는 불 변수
82          int move_x = 0;     // 캐릭터가 이동해야 한다면 현재 위치에서 스크린 상의 이동한 X 좌표 거리를 저장해놓는 용도의 변수
83          int move_y = 0;     // 캐릭터가 이동해야 한다면 현재 위치에서 스크린 상의 이동한 Y 좌표 거리를 저장해놓는 용도의 변수
84          int move_direction = -1;    // 캐릭터가 현재 슬라이딩하고 있는 방향을 나타내는 변수
85
86          /* 게임 플레이와 관련된 데이터들 */
```

19. available_power: 게임에서 아이템을 먹으면 캐릭터의 파워가 올라갑니다. 파워에 비례하여 캐릭터의 속도가 빨라지고, 파워를 소모하여 벽을 하나 부술 수도 있습니다. 이러한 파워의 잔량을 나타냅니다.

20. total_energy: 원래 available_power 의 일부 기능인 캐릭터의 속도를 올려주기 위한 값이었으나, 캐릭터가 에너지를 소모해도 속도가 줄어들지 않는다는 것이 게임을 루즈하게 만드는 것 같아 활용하지 않게 되었습니다. 하지만 누적으로 획득한 아이템 개수는 추후 게임의 진행 방식을 변경할 때 쓰일 수도 있으므로 삭제하지 않고 남겨 두었습니다. (주석은 잘못된 것입니다.)

21. directing: 방향키 중 아무것도 누르지 않고 있으면 -1 의 값을 유지하다가 방향키를 누르고 있으면 해당 방향키 정보를 앞서 정의한 매크로로 저장합니다. 방향키를 떼면 값은 다시 -1 이 됩니다.

22. sliding: 캐릭터가 움직일 때 이미지가 순간이동하는 단순한 구현이 아닌, 부드럽게 방 사이를 이동하도록 구현하였습니다. 이에 슬라이딩 상태라는 것을 정의하였는데, 방향키를 떼서 캐릭터가 해당 방향으로 이동 가능하면 sliding 상태가 되고 부드럽게 이동합니다.

23. move_x, move_y: sliding 상태가 끝나야만 캐릭터의 좌표를 나타내는 값이 실제로 바뀌도록 구현하였기 때문에 sliding 상태 중 캐릭터가 원래의 위치에서 이동한 양을 나타냅니다.

24. move_direction: 캐릭터가 현재 슬라이딩하고 있는 방향을 나타냅니다.

25. initializeGame(): 게임 플레이와 관련된 주요한 변수들의 초기값을 설정하여 데이터 간의 충돌이 일어나지 않게 하는 함수입니다.

26. deinitializeGame(): 게임 플레이와 관련된 주요한 변수들의 값을 초기값으로 복구하여 데이터 간의 충돌이 일어나지 않게 하는 함수입니다.

```
88      ofTrueTypeFont gamename_font;   // 화면에 게임 이름을 출력할 때 사용할 폰트
89      string gamename_string;     // 화면에 출력할 문자열(게임 이름)
90      float gamenameX;    // 문자열을 화면 중앙에 정렬하기 위한 변수
91
92      ofTrueTypeFont gamemode_font;   // 화면에 게임 모드를 출력할 때 사용할 폰트
93      string gamemode_string;     // 화면에 출력할 문자열(게임 모드)
94      float gamemodeX;    // 문자열을 화면 중앙에 정렬하기 위한 변수
95
96      ofTrueTypeFont remain_power_font;   // 화면에 available_power 값을 출력할 때 사용할 폰트
97
98      ofTrueTypeFont result_font;     // 결과 화면에 문자를 출력할 때 사용할 폰트
99      string result_string;   // 화면에 출력할 문자열
100     float resultX;      // 문자열을 화면 중앙에 정렬하기 위한 변수
101
102     ofTrueTypeFont result_font2;    // 결과 화면에 문자를 출력할 때 사용할 폰트
103     string result_string2;  // 화면에 출력할 문자열
104     float resultX2;     // 문자열을 화면 중앙에 정렬하기 위한 변수
105
106     ofxGIF::fiGifLoader happycat;   // 해피캣을 그리기 위한 애드온 자료구조
107     int happycatFrame;  // 해피캣 gif의 프레임을 나타내는 인덱스
108
109     ofImage main_image;     // 메인 화면의 게임 이미지
110     int screen_mode = MAIN; // MAIN_SCREEN or EASY_MODE_SCREEN or HARD_MODE_SCREEN 저장
111
112     ofImage miner_image;    // 게임 캐릭터(광부) 이미지
113     int minerX;             // 미로에서 캐릭터가 위치한 X 좌표
114     int minerY;             // 미로에서 캐릭터가 위치한 Y 좌표
115
116     ofImage item_image;     // 게임 아이템(에너지 드링크) 이미지
117     int itemX;              // 미로에서 아이템이 위치한 X 좌표
118     int itemY;              // 미로에서 아이템이 위치한 Y 좌표
119
120     bool key_lock = false;  // 키 입력을 차단하는 플래그
121     bool easy_mode = false; // 게임이 이지 모드임을 나타내는 플래그
122     bool hard_mode = false; // 게임이 하드 모드임을 나타내는 플래그
```

27. gamename_font, gamename_string, gamenameX: 메인 화면의 게임 제목을 출력하기 위한 폰트, 게임 제목 문자열, 문자열을 가운데 정렬하여 출력하기 위한 텍스트의 X 값입니다.

28. gamemode_font, gamemode_string, gamemodeX: 메인 화면의 게임 모드 및 종료 안내 문자열을 출력하기 위한 폰트, 해당 문자열 및 가운데 정렬하여 출력하기 위한 텍스트의 X 값입니다.

29. remain_power_font: 게임 플레이 화면에서 available_power 의 값을 출력하기 위한 폰트입니다.

30. result_font, result_string, resultX: 결과 화면에 축하말을 출력하기 위한 폰트, 축하말 문자열, 가운데 정렬하여 출력하기 위한 텍스트의 X 값입니다.

31. result_font2, result_string2, resultX2: 결과 화면에 메인 화면으로 돌아가는 방법에 대한 안내말을 출력하기 위한 폰트, 해당 문자열, 가운데 정렬하여 출력하기 위한 텍스트의 X 값입니다.

32. happycat, happycatFrame: 결과 화면에 출력할 고양이 사진과 그 프레임 번호입니다. 고양이 사진이 아래처럼 .gif 확장자의 움직이는 사진이어서 출력을 위해 ofxGIF 애드온을 사용하였습니다.

33. main_image: 메인 화면에 출력할 사진입니다.

34. screen_mode: 현재 화면이 메인 화면인지, 게임 플레이 화면인지, 종료 화면인지에 대한 정보를 저장합니다.

35. miner_image, minerX, minerY: 게임 캐릭터 이미지와 미로 상의 X, Y 좌표입니다. 여기서 X, Y 좌표는 실제 화면 상에 그려지는 위치가 아닌 maze 배열에서의 위치 값입니다.

36. item_image, itemX, itemY: 게임 아이템 이미지와 미로 상의 X, Y 좌표입니다. 여기서 X, Y 좌표는 실제 화면 상에 그려지는 위치가 아닌 maze 배열에서의 위치 값입니다.

37. key_lock: 다른 키 입력을 막아야 할 때 쓰입니다. 대표적인 사용 예시는 sliding 상태일 때 key_lock 을 거는 것입니다.

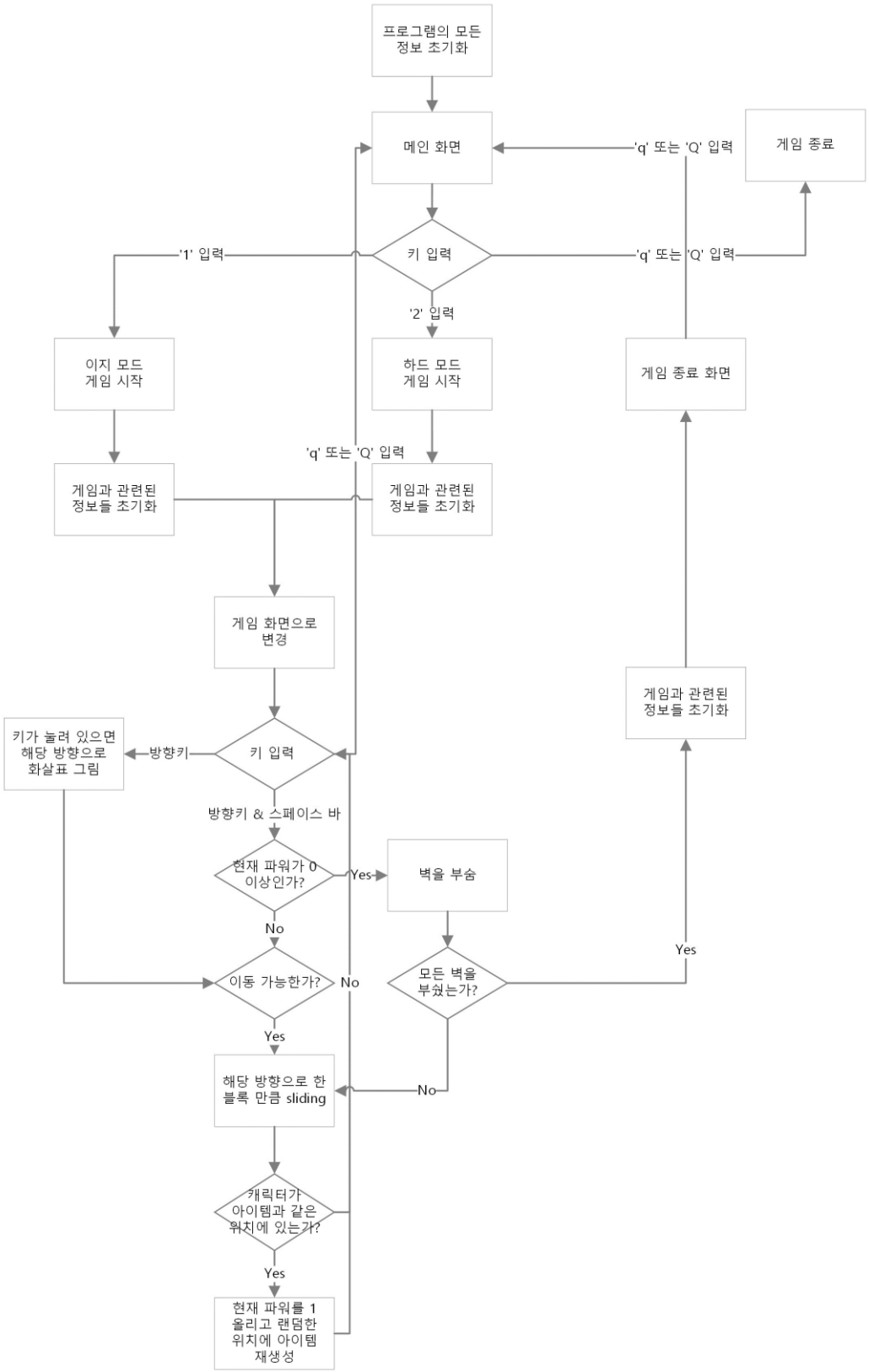
38. easy_mode, hard_mode: 게임 모드가 easy_mode 인지 hard_mode 인지 확인합니다.

3. 프로젝트 전체 플로우차트
+ 자료구조/알고리즘
+ 시간/공간 복잡도

지금부터는 게임의 전체적인 흐름에 대해 요약합니다. 먼저 플로우차트를 통해 대략적으로 어떤 방식으로 게임이 진행되는지 살펴본 뒤 정의된 함수들의 자료구조와 알고리즘을 분석해 보겠습니다.

Openframework 의 대부분의 함수들은 서로 간의 동작이 의존적이기 때문에 코드 상에서 순서대로 한 줄씩 설명하는 것은 이해하기 어렵습니다. 따라서 그 흐름을 파악하기 쉽게끔 함수의 설명 순서를 조금 변경하였습니다.

또한, 제가 구현한 ofApp 클래스의 기본적인 몇몇 함수들은 자료구조/알고리즘과 시간/공간 복잡도 분석이 의미가 없습니다. 따라서 그 분석이 의미가 있는 경우에만 설명을 제시하겠습니다.



위는 프로그램의 전체적인 플로우차트입니다. 흐름을 이해할 수 있도록 변수명과 함수명을 생략하고 그 동작만을 기술하였습니다. 이어서 사용된 자료구조와 각 함수의 알고리즘을 분석하며 프로그램의 구체적인 동작에 대해 설명해 보겠습니다.

```cpp
void ofApp::setup(){
    srand(time(0));     // 랜덤 미로 생성을 위해 시드 값 설정

    ofSetVerticalSync(true);    // 수직 동기화로 모니터 주사율에 맞추어 초당 출력 프레임 설정
    ofBackground(ofColor::white);   // 배경 흰색으로 설정

    /* 폰트 출처 : https://maplestory.nexon.com/Media/Font */
    gamename_font.load("MaplestoryBold.ttf", 50);       // 게임 이름을 쓰기 위한 폰트 크기 50으로 설정
    gamemode_font.load("MaplestoryBold.ttf", 20);       // 게임 모드를 쓰기 위한 폰트 크기 30으로 설정
    remain_power_font.load("MaplestoryBold.ttf", 15);   // available_power를 나타내기 위한 폰트 크기 15로 설정
    result_font.load("MaplestoryBold.ttf", 75);         // 결과 화면에 쓰기 위한 폰트 크기 75로 설정
    result_font2.load("MaplestoryBold.ttf", 30);        // 결과 화면에 쓰기 위한 폰트 크기 60으로 설정

    gamename_string = "Break the Maze";                                 // 화면에 출력할 게임 이름 문자열
    gamemode_string = "1. EASY MODE(5 X 5)  2. HARD MODE(10 X 10)   Q. EXIT";   // 화면에 출력할 게임 모드 문자열

    result_string = "GAME COMPLETED";   // 결과 화면 상단에 출력할 게임 완료 문자열
    resultX = result_font.stringWidth(result_string);
    resultX = (ofGetWidth() - resultX) / 2; // 텍스트를 화면 중앙에 정렬하기 위한 계산

    result_string2 = "Press 'q' or 'Q' to return to the main screen";   // 결과 화면에 출력할 안내 문자열
    resultX2 = result_font2.stringWidth(result_string2);
    resultX2 = (ofGetWidth() - resultX2) / 2;   // 텍스트를 화면 중앙에 정렬하기 위한 계산

    gamenameX = gamename_font.stringWidth(gamename_string);
    gamenameX = (ofGetWidth() - gamenameX) / 2;     // 텍스트를 화면 중앙에 정렬하기 위한 계산

    gamemodeX = gamemode_font.stringWidth(gamemode_string);
    gamemodeX = (ofGetWidth() - gamemodeX) / 2;     // 텍스트를 화면 중앙에 정렬하기 위한 계산

    main_image.load("images/main.png");     // 메인 이미지 로드
    miner_image.load("images/miner.png");   // 캐릭터 이미지 로드
    item_image.load("images/item.png");     // 아이템 이미지 로드

    happycat.load("images/happycat.gif");   // 해피캣 이미지 로드
}
```

프로그램의 모든 정보는 ofApp::setup()에서 초기화됩니다. 이 함수의 동작은 다음과 같습니다.

1. 프로그램 실행 시마다 랜덤한 미로를 생성하기 위해 시드 값을 설정
2. 수직 동기화를 사용하여 모니터 주사율과 초당 출력 프레임을 동기화
3. 배경 색을 흰색으로 설정
4. 사용되는 모든 폰트와 그 문자열을 설정하고 가운데 정렬하기 위해 X 좌표 값 계산
5. 화면에 출력할 모든 이미지 로드

```
109 void ofApp::draw(){
110     ofSetColor(ofColor::darkGray);  // 화면 테두리 색깔
111     ofDrawRectangle(0, 0, 1024, 50);    // 화면 상단 테두리
112     ofDrawRectangle(0, 0, 50, 768); // 화면 좌측 테두리
113     ofDrawRectangle(0, 718, 1024, 50);  // 화면 하단 테두리
114     ofDrawRectangle(974, 0, 50, 768);   // 화면 우측 테두리
115 
116 
117     if (screen_mode == MAIN) {  // 메인 화면 그리기
118         ofSetColor(255);    // 이미지 출력을 위한 색 설정
119         /* 메인 화면 이미지 출력 */
120         main_image.draw(50, 90, 924, 588);
121 
122         ofSetColor(ofColor::black); // 문자열은 검정색으로 출력
123 
124         /* 게임 이름 문자열 출력 */
125         gamename_font.drawString(gamename_string, gamenameX, 105);  // 화면 중앙에 정렬하여 게임 이름 출력
126 
127         /* 게임 모드 문자열 출력 */
128         gamemode_font.drawString(gamemode_string, gamemodeX, 710);  // 화면 중앙에 정렬하여 게임 모드 출력
129 
130     }
131     else if (screen_mode == EASY_MODE || screen_mode == HARD_MODE) {        // 이지 모드 또는 하드 모드일 시
132         drawGameScreen();       // 게임 플레이 화면을 그린다
133     }
134     else if (screen_mode == RESULT) {
135         ofSetColor(ofColor::lightGoldenRodYellow);  // 화면 테두리 색깔
136         ofDrawRectangle(0, 0, 1024, 50);    // 화면 상단 테두리
137         ofDrawRectangle(0, 0, 50, 768); // 화면 좌측 테두리
138         ofDrawRectangle(0, 718, 1024, 50);  // 화면 하단 테두리
139         ofDrawRectangle(974, 0, 50, 768);   // 화면 우측 테두리
140 
141         ofSetColor(255);
142         for (int i = 1; i <= 17; i++) {
143             for (int j = 0; j <= 5; j++)
144                 happycat.pages[happycatFrame].draw(50 * i, 50 + 100 * j);
145         }
146 
147         ofSetColor(ofColor::black);
148         result_font.drawString(result_string, resultX, 120);
149         result_font2.drawString(result_string2, resultX2, 618);
150     }
151 }
```

게임 화면은 ofApp::draw()에서 screen_mode 에 저장된 값을 바탕으로 계속해서 그 프레임을 하나씩 그립니다. 모든 화면에 공통적으로 darkGray 색의 테두리가 먼저 그려지고, 메인 화면과 결과 화면은 간단하게 그릴 수 있기 때문에 draw() 안에서 직접 그려집니다.

메인 화면을 그리는 것은 게임 메인 이미지 출력, 게임 이름 문자열과 게임 모드 출력으로 끝이고, 결과 화면을 그리는 것은 테두리 색깔을 lightGoldenRodYellow 색으로 변경하고 happycat 이미지를 여러 개 출력한 후 축하말, 안내말을 출력하는 것으로 끝입니다.

screen_mode 가 게임 모드일 경우 공통되게 drawGameScreen()을 호출합니다.

```
154 void ofApp::keyPressed(int key){
155
156     if (!key_lock) {
157
158         if (key == 'q' || key == 'Q') { // q/Q 입력을 받으면
159             switch (screen_mode) {      // 현재 스크린 상태에 따라 다르게 처리한다
160             case MAIN:  // 메인 화면이면
161                 ofExit();       // 게임 종료
162             case EASY_MODE: // 게임 플레이 화면이면
163                 deinitializeMaze();     // 미로와 관련된 정보 제거
164                 deinitializeGame();     // 게임과 관련된 정보 제거
165                 screen_mode = MAIN;     // 메인 화면으로 변경
166                 easy_mode = false;      // 이지 모드임을 나타내는 플래그 비활성화
167                 break;
168             case HARD_MODE: // 게임 플레이 화면이면
169                 deinitializeMaze();     // 미로와 관련된 정보 제거
170                 deinitializeGame();     // 게임과 관련된 정보 제거
171                 screen_mode = MAIN;     // 메인 화면으로 변경
172                 hard_mode = false;      // 하드 모드임을 나타내는 플래그 비활성화
173                 break;
174             case RESULT:    // 결과 화면이면
175                 screen_mode = MAIN;     // 메인 화면으로 변경
176                 break;
177             }
178
179             return;
180         }
181
182         if (key == '1' && screen_mode == MAIN) {    // '1'을 누르면 이지 모드로 게임 시작
183             initializeMaze(EASY_MODE);  // 미로 정보 초기화
184             initializeGame();   // 게임과 관련된 정보 초기화
185             easy_mode = true;   // 이지 모드임을 나타내는 플래그 활성화
186             screen_mode = EASY_MODE;    // 스크린을 이지 모드로 변경
187
188             return;
189
190         }
191
192         if (key == '2' && screen_mode == MAIN) {
193             initializeMaze(HARD_MODE);
194             initializeGame();   // 게임과 관련된 정보 초기화
195             hard_mode = true;   // 하드 모드임을 나타내는 플래그 활성화
196             screen_mode = HARD_MODE;    // 스크린을 하드 모드로 변경
197
198             return;
199         }
```

```
198         if (key == OF_KEY_LEFT && (easy_mode || hard_mode)) {   // 게임 중 왼쪽 방향키를 누르고 있으면
199             directing = LEFT;   // 왼쪽을 향하고 있다고 표시
200             return;
201         }
202         if (key == OF_KEY_RIGHT && (easy_mode || hard_mode)) {  // 게임 중 오른쪽 방향키를 누르고 있으면
203             directing = RIGHT;  // 오른쪽을 향하고 있다고 표시
204             return;
205         }
206         if (key == OF_KEY_UP && (easy_mode || hard_mode)) {     // 게임 중 위쪽 방향키를 누르고 있으면
207             directing = UP;     // 위를 향하고 있다고 표시
208             return;
209         }
210         if (key == OF_KEY_DOWN && (easy_mode || hard_mode)) {   // 게임 중 아래쪽 방향키를 누르고 있으면
211             directing = DOWN;   // 아래를 향하고 있다고 표시
212             return;
213         }
```

```
215         if (key == ' ' && directing != -1 && available_power > 0) { // 스페이스 바를 누르는 순간 키보드로 어딘가 향하고 있으며, available_power가 1 이상인 경우
216             switch (directing) {
217             case LEFT:
218                 if (maze[minerY][minerX - 1] == '¦') {  // 좌측에 있는 게 세로 벽이면
219                     maze[minerY][minerX - 1] = ' ';     // 벽을 허물고
220                     total_wall--;   // 벽의 개수를 1 감소한 후
221                     available_power--;  // available_power를 1 감소한다
222                 }
223                 break;
224             case RIGHT:
225                 if (maze[minerY][minerX + 1] == '¦') {  // 우측에 있는 게 세로 벽이면
226                     maze[minerY][minerX + 1] = ' ';     // 벽을 허물고
227                     total_wall--;   // 벽의 개수를 1 감소한 후
228                     available_power--;  // available_power를 1 감소한다
229                 }
230                 break;
231             case UP:
232                 if (maze[minerY - 1][minerX] == '-') {  // 위에 있는 게 가로 벽이면
233                     maze[minerY - 1][minerX] = ' ';     // 벽을 허물고
234                     total_wall--;   // 벽의 개수를 1 감소한 후
235                     available_power--;  // available_power를 1 감소한다
236                 }
237                 break;
238             case DOWN:
239                 if (maze[minerY + 1][minerX] == '-') {  // 아래에 있는 게 세로 벽이면
240                     maze[minerY + 1][minerX] = ' ';     // 벽을 허물고
241                     total_wall--;   // 벽의 개수를 1 감소한 후
242                     available_power--;  // available_power를 1 감소한다
243                 }
244                 break;
245             }
246 
247             if (total_wall == 0) {  // 게임 완료 조건: 모든 벽을 허문 경우
248                 deinitializeGame();     // 정보 초기화
249                 deinitializeMaze();     // 정보 초기화
250                 screen_mode = RESULT;   // 결과 화면으로 전환
251                 easy_mode = false;
252                 hard_mode = false;
253             }
254         }
255 
256     }
257 }
```

ofApp::KeyPressed()에서는 키를 누르거나, 누르는 중의 동작을 정의합니다. key_lock 이 걸려 있으면 키 입력을 무시합니다.

1. q/Q 입력을 받았을 시: 메인 화면이면 프로그램을 종료합니다. 게임 플레이 화면이면 미로, 게임과 관련된 정보들을 초기화한 후 메인 화면으로 변경하고 게임 모드와 관련된 플래그를 reset 합니다. 결과 화면이면 메인 화면으로 변경합니다.
2. 1 입력을 받았고, 현재 screen_mode 가 메인 화면일 시: 미로, 게임과 관련된 정보들을 초기화한 후 이지 모드를 나타내는 플래그를 set 합니다.
3. 2 입력을 받았고, 현재 screen_mode 가 메인 화면일 시: 미로, 게임과 관련된 정보들을 초기화한 후 하드 모드를 나타내는 플래그를 set 합니다.
4. 방향키 입력을 받았고, 현재 게임 모드가 활성화돼있을 시: 캐릭터가 해당 방향키 방향을 향하고 있다고 directing 변수에 LEFT/RIGHT/UP/DOWN 중 하나의 매크로 값을 저장하여 표시합니다.
5. 스페이스 바 입력을 받았고, 현재 directing 에 방향 값이 저장되어 있고, available_power 가 0 보다 클 시: 벽을 부술 수 있는 조건으로, 현재 캐릭터의 좌표로부터 캐릭터가 향하고 있는 방향으로 maze 상에서 1 만큼 떨어진 위치에 벽이 존재하는지 판단합니다. 벽이 존재하면 공백 문자를 저장하여 벽을 없애고 total_wall 과 available_power 를 1 감소시킵니다. 이때 total_wall 의 값이 0 이 되면 모든 벽이 사라졌다는 의미이므로 미로, 게임과 관련된 정보를 초기화하고 screen_mode 를 결과 화면의 값으로 바꾼 뒤 게임 모드 flag 를 reset 합니다.

```cpp
void ofApp::keyReleased(int key){
    if (!key_lock) {
        if (key == OF_KEY_LEFT && (easy_mode || hard_mode)) {   // 왼쪽 방향키를 떼면
            directing = -1;     // 아무 방향도 향하지 않고 있다고 표시하고
            if (maze[minerY][minerX - 1] == ' ') {  // 만약 벽이 존재하지 않으면
                //minerX -= 2;  // 이동 가능하면 캐릭터를 이동
                move_x = -(step / 10) - (2 * available_power);  // 좌측 방향으로 슬라이딩하기 위해 miner의 x축 이동 거리를 -(step / 10) - (2 * available_power)로 설정
                move_y = 0;     // y축 이동 거리는 0이 됨
                move_direction = LEFT;      // 좌측으로 슬라이딩한다는 표시, update()에 알림
                key_lock = true;    // 다른 키 입력을 무시
                sliding = true;     // 슬라이딩 중임을 나타냄
            }
            return;
        }

        if (key == OF_KEY_RIGHT && (easy_mode || hard_mode)) {  // 오른쪽 방향키를 떼면
            directing = -1;     // 아무 방향도 향하지 않고 있다고 표시하고
            if (maze[minerY][minerX + 1] == ' ') {  // 만약 벽이 존재하지 않으면
                //minerX += 2;  // 이동 가능하면 캐릭터를 이동
                move_x = (step / 10) + (2 * available_power);       // 우측 방향으로 슬라이딩하기 위해 miner의 x축 이동 거리를 (step / 10) + (2 * available_power)로 설정
                move_y = 0;     // y축 이동 거리는 0이 됨
                move_direction = RIGHT;     // 우측으로 슬라이딩한다는 표시, update()에 알림
                key_lock = true;    // 다른 키 입력을 무시
                sliding = true;     // 슬라이딩 중임을 나타냄
            }
            return;
        }

        if (key == OF_KEY_UP && (easy_mode || hard_mode)) {     // 위쪽 방향키를 떼면
            directing = -1;     // 아무 방향도 향하지 않고 있다고 표시하고
            if (maze[minerY - 1][minerX] == ' ') {
                //minerY -= 2;  // 이동 가능하면 캐릭터를 이동
                move_x = 0;     // x축 이동 거리는 0이 됨
                move_y = -(step / 10) - (2 * available_power);  // 위 방향으로 슬라이딩하기 위해 miner의 y축 이동 거리를 -(step / 10) - (2 * available_power)로 설정
                move_direction = UP;    // 위로 슬라이딩한다는 표시, update()에 알림
                key_lock = true;    // 다른 키 입력을 무시
                sliding = true;     // 슬라이딩 중임을 나타냄
            }
            return;
        }

        if (key == OF_KEY_DOWN && (easy_mode || hard_mode)) {   // 아래쪽 방향키를 떼면
            directing = -1;     // 아무 방향도 향하지 않고 있다고 표시하고
            if (maze[minerY + 1][minerX] == ' ') {
                //minerY += 2;  // 이동 가능하면 캐릭터를 이동
                move_x = 0;     // x축 이동 거리는 0이 됨
                move_y = (step / 10) + (2 * available_power);       // 아래 방향으로 슬라이딩하기 위해 miner의 y축 이동 거리를 (step / 10) + (2 * available_power)로 설정
                move_direction = DOWN;  // 아래로 슬라이딩한다는 표시, update()에 알림
                key_lock = true;    // 다른 키 입력을 무시
                sliding = true;     // 슬라이딩 중임을 나타냄
            }
            return;
        }
    }
}
```

ofApp::keyReleased()에서는 키를 뗐을 때의 동작을 정의합니다. key_lock 이 걸려 있으면 키 입력을 무시합니다. 이 함수에는 방향키를 뗐을 때의 동작만 정의하고 있는데 이 부분이 중요합니다.

방향키를 누르고 있으면 해당 방향을 향하고 있다는 의미이므로 directing 에 미리 정의한 매크로 방향 값을 저장하였습니다. 방향키를 떼면 먼저 directing 을 다시 초기값으로 되돌림으로써 아무 방향도 향하지 않고 있음을 나타냅니다.

그리고 미로에서 캐릭터가 위치한 방향으로부터 기존에 향하고 있던 방향으로 1 만큼 떨어진 위치의 값이 ' '으로 벽이 존재하지 않을 경우 앞서 설명한 sliding 동작이 수행됩니다. 먼저 move_x, move_y 의 초기 값을 설정합니다. 이는 화면 상에서 캐릭터가 sliding 중에 프레임마다 이동할 거리의 픽셀 값을 나타냅니다. 그리고 move_direction 에 기존의 directing 이 가리키던 값이 저장되어 sliding 방향을 나타낼 것입니다. key_lock 을 set 하여 sliding 이 끝나기 전까지 다른 키 입력은 무시하고, sliding 도 set 하여 현재 sliding 중임을 나타냅니다.

```cpp
void ofApp::update(){
    if (screen_mode == RESULT && ofGetElapsedTimeMillis() % 3) {
        happycatFrame++;
        if (happycatFrame > happycat.pages.size() - 1) happycatFrame = 0;
    }

    if (sliding && (easy_mode || hard_mode)) {

        if (move_direction == LEFT) {   // 왼쪽으로 슬라이딩 중인 경우
            move_x -= (step / 10) + (2 * available_power);  // miner를 왼쪽으로 (step / 10) + (2 * available_power) 픽셀 슬라이딩함
            if (LEFT_X + (minerX * step) + move_x <= LEFT_X + ((minerX - 2) * step)) {  // (현재 실제 위치) + move_x <= (미래의 위치)
                key_lock = false;       // 키 잠금 해제
                move_direction = -1;    // 슬라이딩 방향 초기화
                minerX -= 2;            // miner의 좌표 실제로 이동
                move_x = 0;             // move_x 초기화
                sliding = false;        // sliding 초기화
            }
        }
        else if (move_direction == RIGHT) { // 오른쪽으로 슬라이딩 중인 경우
            move_x += (step / 10) + (2 * available_power);  // miner를 오른쪽으로 (step / 10) + (2 * available_power)픽셀 슬라이딩함
            if (LEFT_X + (minerX * step) + move_x >= LEFT_X + ((minerX + 2) * step)) {  // (현재 실제 위치) + move_x >= (미래의 위치)
                key_lock = false;       // 키 잠금 해제
                move_direction = -1;    // 슬라이딩 방향 초기화
                minerX += 2;            // miner의 좌표 실제로 이동
                move_x = 0;             // move_x 초기화
                sliding = false;        // sliding 초기화
            }
        }
        else if (move_direction == UP) {    // 위로 슬라이딩 중인 경우
            move_y -= (step / 10) + (2 * available_power);
            if (TOP_Y + (minerY * step) + move_y <= TOP_Y + ((minerY - 2) * step)) {    // (현재 실제 위치) + move_y <= (미래의 위치)
                key_lock = false;       // 키 잠금 해제
                move_direction = -1;    // 슬라이딩 방향 초기화
                minerY -= 2;            // miner의 좌표 실제로 이동
                move_y = 0;             // move_y 초기화
                sliding = false;        // sliding 초기화
            }
        }
        else if (move_direction == DOWN) {
            move_y += (step / 10) + (2 * available_power);
            if (TOP_Y + (minerY * step) + move_y >= TOP_Y + ((minerY + 2) * step)) {    // (현재 실제 위치) + move_y >= (미래의 위치)
                key_lock = false;       // 키 잠금 해제
                move_direction = -1;    // 슬라이딩 방향 초기화
                minerY += 2;            // miner의 좌표 실제로 이동
                move_y = 0;             // move_y 초기화
                sliding = false;        // sliding 초기화
            }
        }

        if (!sliding) { // 위 조건문을 통과한 후 sliding이 false가 되었다면 위치의 갱신이 이루어진 것이다
            if (minerX == itemX && minerY == itemY) {   // 캐릭터가 아이템을 획득하면
                total_energy++;     // 현재까지 획득한 에너지 개수 1 증가
                available_power++;  // 현재 갖고 있는 벽 부수기 횟수 1 증가

                do {    // 랜덤한 위치에 아이템 다시 생성
                    itemY = rand() % maze_row;  // 아이템의 행 위치 생성
                    itemX = rand() % maze_col;  // 아이템의 열 위치 생성
                } while (itemY % 2 == 0 ||  // 2로 나누어 떨어지는 위치엔 벽이 있음
                    itemX % 2 == 0 ||   // 2로 나누어 떨어지는 위치엔 벽이 있음
                    (itemY == minerY && itemX == minerX));  // 캐릭터와 완전히 위치가 겹칠 수 없음
            }
        }
    }

}
```

ofApp::update()에서는 프레임마다 변경돼야 하는 내용을 정의합니다.

결과 화면이면 출력할 고양이 사진의 프레임을 순환적으로 변경합니다.

sliding 중이면 move_x 또는 move_y 의 값을 해당 방향으로 이동할 수 있도록 일정 값 더하거나 빼고, 만약 화면 상에서 캐릭터가 sliding 이 끝난 위치에 도달하면 key_lock 을 해제하고 move_direction 을 초기값으로 되돌린 후 maze 벡터 상에서의 캐릭터의 위치를 나타내는 minerX 또는 minerY 의 값을 조정합니다. 그리고 sliding flag 를 reset 합니다. 이후의 if(!sliding) 조건문이 수행된다는 것은 앞선 if-else 조건문에서 sliding 이 끝났다는 의미이므로 sliding 이 끝난 후 캐릭터와 아이템의 위치가 일치하는지 비교합니다. 만약

캐릭터와 아이템의 위치가 일치한다면 total_energy, available_power 를 1 증가시키고 아이템의 위치 값을 다시 캐릭터와 겹치지 않는 랜덤한 위치로 생성합니다.

```cpp
void ofApp::drawGameScreen() {
    /* 미로의 테두리를 그린다 */
    ofSetColor(ofColor::black);     // 선 색깔은 검정색으로 설정
    ofSetLineWidth(10);             // 선 굵기는 10으로 설정

    ofDrawLine(LEFT_X - MARGIN, TOP_Y, RIGHT_X + MARGIN, TOP_Y);          // 상단 테두리
    ofDrawLine(LEFT_X, TOP_Y - MARGIN, LEFT_X, BOTTOM_Y + MARGIN);      // 좌측 테두리
    ofDrawLine(LEFT_X - MARGIN, BOTTOM_Y, RIGHT_X + MARGIN, BOTTOM_Y);  // 하단 테두리
    ofDrawLine(RIGHT_X, TOP_Y - MARGIN, RIGHT_X, BOTTOM_Y + MARGIN);       // 우측 테두리

    /* 미로의 벽들을 그린다 */
    step = MAZE_LENGTH / (maze_col - 1);    // 매우 중요한 값: 화면 상에서 미로가 그려지기 시작하는 (LEFT_X, TOP_Y)로부터 maze의 각 원소가 얼마나 떨어져 있는지 나타낸다.
    // ex: maze[r][c]의 화면 상의 위치는 (LEFT_X + (step * c), TOP_Y + (step * r))
    // ex: 한 행의 방이 5개면 미로의 열은 11개이고, 왼쪽 모서리에서 오른쪽 모서리까지 10번 건너 뛰어야 한다.

    for (int r = 0, penY = TOP_Y, penX = LEFT_X; r < maze_row - 1; r++, penX = LEFT_X, penY += step) {  // penX: 선을 그릴 커서가 위치하는 X 좌표, penY: 선을 그릴 커서가 위치하는 Y 좌표
        for (int c = 0; c < maze_col - 1; c++, penX += step) {
            if (maze[r][c] == '-') ofDrawLine(penX - step - MARGIN, penY, penX + step + MARGIN, penY);  // 가로 벽 그리기
            if (maze[r][c] == '|') ofDrawLine(penX, penY - step - MARGIN, penX, penY + step + MARGIN);  // 세로 벽 그리기
        }
    }

    /* 게임의 entities(character, item, available_power)를 그린다 */
    ofSetColor(255);    // 이미지를 그리기 위해 색을 255으로 설정
    image_size = step + (4 * MARGIN);   // 그릴 entity의 이미지 너비 및 높이

    item_image.draw(LEFT_X + (itemX * step) - (image_size / 2), TOP_Y + (itemY * step) - (image_size / 2), image_size, image_size);     // 아이템을 그린다
    if (!sliding) {     // 캐릭터가 슬라이딩 중이 아니면
        miner_image.draw(LEFT_X + (minerX * step) - (image_size / 2), TOP_Y + (minerY * step) - (image_size / 2), image_size, image_size);  // 캐릭터를 좌표에 즉시 그린다
    }
    else {      // 캐릭터가 슬라이딩 중이면
        miner_image.draw(LEFT_X + (minerX * step) - (image_size / 2) + move_x, TOP_Y + (minerY * step) - (image_size / 2) + move_y, image_size, image_size);    // 캐릭터를 이전 좌표에 move_x, move_y를 추가한 중간 지점에 그린다
    }
    ofSetColor(ofColor::red);
    remain_power_font.drawString("POWER: " + ofToString(available_power), RIGHT_X + 15, BOTTOM_Y - 15);     // 실시간으로 available_power의 값을 출력한다

    if (directing == -1) return;
    else if (directing == LEFT) ofDrawArrow(ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step)), ofVec3f(LEFT_X + (minerX * step) - (MARGIN * 10), TOP_Y + (minerY * step)), 10);    // 캐릭터가 향하고 있는 방향으로 화살표를 그린다
    else if (directing == RIGHT) ofDrawArrow(ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step)), ofVec3f(LEFT_X + (minerX * step) + (MARGIN * 10), TOP_Y + (minerY * step)), 10);   // 캐릭터가 향하고 있는 방향으로 화살표를 그린다
    else if (directing == UP) ofDrawArrow(ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step)), ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step) - (MARGIN * 10)), 10);     // 캐릭터가 향하고 있는 방향으로 화살표를 그린다
    else if (directing == DOWN) ofDrawArrow(ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step)), ofVec3f(LEFT_X + (minerX * step), TOP_Y + (minerY * step) + (MARGIN * 10)), 10);   // 캐릭터가 향하고 있는 방향으로 화살표를 그린다
}
```

ofApp::drawGameScreen()은 ofApp::draw()에서 screen_mode 가 게임 화면을 나타내는 값일 때 프레임마다 호출되는 함수입니다. 지금부터 설명할 몇몇 함수들은 그 자료구조와 알고리즘을 요약하고 효율성을 분석하는 과정이 필요합니다. 이 함수의 알고리즘은 다음과 같습니다.

1. (LEFT_X, TOP_Y), (RIGHT_X, TOP_Y), (RIGHT_X, BOTTOM_Y), (LEFT_X, BOTTOM_Y)를 잇는 미로의 외곽선(테두리)을 그립니다.
2. 앞서 설명한 step 의 값을 미로의 너비(또는 높이)를 (maze 의 column 개수) − 1 으로 나눈 값으로 설정합니다. 이는 주석에 예시를 들어둔 것과 같이, 한 행의 방이 5 개라면 maze 의 column 은 11 개이고 왼쪽 모서리에서부터 오른쪽 모서리까지는 10 번의 step 을 건너 뛰어야 하기 때문입니다.
3. 이중 for loop 로 maze 의 모든 원소를 확인하며 벽인지 판단하고 벽을 그리는 작업을 진행합니다.
    A. maze 의 행을 나타내는 r, 열을 나타내는 c 가 정의됩니다.
    B. 실제로 화면에서 maze 의 각 요소를 그리기 시작할 점의 위치를 나타내는 penX, penY 가 정의됩니다. 각각의 초기값은 좌상단 모서리의 화면 상의 위치이며, 매 반복마다 step 만큼 증가합니다.
    C. maze[r][c]가 가로 벽이면 (penX, penY)에서 좌우로 (step + MARGIN)만큼의 선을 그립니다.
    D. maze[r][c]가 세로 벽이면 (penX, penY)에서 상하로 (step + MARGIN)만큼의 선을 그립니다.
4. image_size 에 화면에 그릴 캐릭터, 아이템 이미지의 한 변의 길이를 저장합니다.
5. (LEFT_X + itemX * step, TOP_Y + itemY * step) 좌표를 중심 좌표로 갖고 image_size X image_size 의 크기를 갖는 아이템 이미지를 그립니다.

6. 캐릭터가 슬라이딩 중이 아니면 (LEFT_X + minerX * step, TOP_Y + minerY * step) 좌표를 중심 좌표로 갖고 image_size X image_size 의 크기를 갖는 캐릭터 이미지를 그립니다. 캐릭터가 sliding 중이면 X 좌표와 Y 좌표 값에 move_x, move_y 값을 더하여 sliding 모션을 구현합니다.
7. 현재 available_power 의 값을 폰트로 출력합니다.
8. 방향키를 누르고 있는 중이라면 directing 은 -1 이 아닌 값을 갖게 될 것이고, 이 경우 directing 의 값에 따라 해당 방향으로의 화살표를 그립니다.

이 함수에서는 2 차원 벡터 maze 의 모든 원소를 확인하면서 벽인지 아닌지 검사하고, 벽을 그리는 3 번의 이중 for loop 가 시간 복잡도와 공간 복잡도에 있어 지배적인 영향을 가집니다. 미로는 정사각형 형태로 행과 열의 개수가 갖으므로 그 값을 N 이라고 한다면 시간 복잡도는 O(N^2)이 되고, 공간 복잡도 역시 O(N^2)이 될 것입니다. 나머지 동작들은 모두 상수 시간의 시간 복잡도를 갖습니다.

```cpp
void ofApp::initializeMaze(int game_mode) {

    if (game_mode == EASY_MODE) {
        HEIGHT = 5;     // 미로의 높이(방 개수) 설정
        WIDTH = 5;      // 미로의 너비(방 개수) 설정

        maze_row = HEIGHT * 2 + 1;  // maze 벡터의 행 개수 유도
        maze_col = WIDTH * 2 + 1;   // maze 벡터의 열 개수 유도

        maze.resize(maze_row, vector<char>(maze_col, '+'));     // 미로의 모든 원소를 모서리('+')로 초기화

        /* 외벽은 부술 수 없으므로 '+'로 유지, 외벽을 제외한 원소를 방(' '), 가로 벽('-'), 세로 벽('¦')으로 초기화 */
        for (int r = 1; r < maze_row - 1; r++) {    // 외벽의 행 인덱스는 0 또는 maze_row - 1이므로 이를 제외하고 반복
            for (int c = 1; c < maze_col - 1; c++) {    // 외벽의 열 인덱스는 0 또는 maze_col - 1이므로 이를 제외하고 반복
                if (r % 2 == 0 && c % 2 == 0) maze[r][c] = ' '; // 방(' ') 조건
                else if (r % 2 == 0 && c % 2 == 1) {
                    maze[r][c] = '¦';    // 세로 벽('¦') 조건
                    total_wall++;   // 미로 내의 벽 개수를 1 증가
                }

                else if (r % 2 == 1 && c % 2 == 0) {
                    maze[r][c] = '-';    // 가로 벽('-') 조건
                    total_wall++;   // 미로 내의 벽 개수를 1 증가
                }
            }
        }

        visited.resize(maze_row, vector< bool >(maze_col, false));  // 방문 여부를 나타내는 2차원 벡터 초기화
        createMaze(1, 1);   // 재귀적으로 완전 미로 구성

        return;
    }

    if (game_mode == HARD_MODE) {
        HEIGHT = 10;    // 미로의 높이(방 개수) 설정
        WIDTH = 10;     // 미로의 너비(방 개수) 설정

        maze_row = HEIGHT * 2 + 1;  // maze 벡터의 행 개수 유도
        maze_col = WIDTH * 2 + 1;   // maze 벡터의 열 개수 유도

        maze.resize(maze_row, vector<char>(maze_col, '+'));     // 미로의 모든 원소를 모서리('+')로 초기화

        /* 외벽은 부술 수 없으므로 '+'로 유지, 외벽을 제외한 원소를 방(' '), 가로 벽('-'), 세로 벽('¦')으로 초기화 */
        for (int r = 1; r < maze_row - 1; r++) {    // 외벽의 행 인덱스는 0 또는 maze_row - 1이므로 이를 제외하고 반복
            for (int c = 1; c < maze_col - 1; c++) {    // 외벽의 열 인덱스는 0 또는 maze_col - 1이므로 이를 제외하고 반복
                if (r % 2 == 0 && c % 2 == 0) maze[r][c] = ' '; // 방(' ') 조건

                else if (r % 2 == 0 && c % 2 == 1) {
                    maze[r][c] = '¦';    // 세로 벽('¦') 조건
                    total_wall++;   // 미로 내의 벽 개수를 1 증가
                }

                else if (r % 2 == 1 && c % 2 == 0) {
                    maze[r][c] = '-';    // 가로 벽('-') 조건
                    total_wall++;   // 미로 내의 벽 개수를 1 증가
                }
            }
        }

        visited.resize(maze_row, vector< bool >(maze_col, false));  // 방문 여부를 나타내는 2차원 벡터 초기화
        createMaze(1, 1);   // 재귀적으로 완전 미로 구성
        return;
    }
```

ofApp::initializeMaze()는 메인 화면에서 게임 플레이 화면으로 전환될 때 미로의 기본적인 정보들을 초기화하는 함수입니다. 이 함수는 이지 모드와 하드 모드에 대해 유사한 알고리즘을 가지므로 공통된 알고리즘을 설명해 보겠습니다.

1. HEIGHT, WIDTH 에 미로의 행, 열에 존재하는 방의 개수를 저장합니다.
2. HEIGHT, WIDTH 로부터 maze_row, maze_col 의 개수를 유도합니다.
3. 2 차원 벡터 maze 의 크기를 미로의 모든 방, 벽, 외곽선을 포함하도록 조정합니다.
4. 이중 for loop 를 돌면서 어떤 벽도 허물어지지 않은 미로의 초기 상태를 구현합니다. 이때 미로의 외곽선은 초기화하지 않기 위해 offset 을 1 부터 maze_row(또는 maze_col) – 2 까지만 조정합니다.
    A. 미로의 행과 열 값이 둘 다 홀수라면 방입니다. 해당 위치에 ‘ ‘을 저장합니다.
    B. 미로의 행은 홀수, 열은 짝수라면 세로 벽입니다. total_wall 을 1 증가하고 해당 위치에 ‘¦’을 저장합니다.

C. 미로의 행은 짝수, 열은 홀수라면 가로 벽입니다. total_wall 을 1 증가하고 해당 위치에 '-'을 저장합니다.

5. 2 차원 벡터 visited 의 크기를 maze 의 크기로 조정하고, 모든 원소를 false 로 초기화합니다.

6. createMaze(1, 1)를 호출하여 maze 의 (1, 1) 좌표부터 시작하여 완전 미로를 구성합니다.

createMaze() 부분을 제외하고, 이 함수에서 시간 복잡도와 공간 복잡도에 있어 지배적인 영향을 미치는 것은 4 번의 이중 for loop 입니다. 미로의 모든 요소를 검토하면서 값을 수정하기 때문에 시간 복잡도와 공간 복잡도는 이전의 경우와 같이 O(N^2)가 됩니다.

```cpp
void ofApp::createMaze(int row, int col) {

    visited[row][col] = true;   // 이번 노드가 방문되었음을 표시

    int direction[4] = { UP, DOWN, LEFT, RIGHT };   // 0: 상, 1: 하, 2: 좌, 3: 우 ¦ 랜덤 미로 생성을 위한 createMaze() 함수에서 사용될 방향 정보
    random_shuffle(direction, direction + 4);   // 다음에 방문할 방향의 순서 무작위로 섞기

    for (int option = 0; option < 4; ++option) {    // direction의 인덱스

        switch (direction[option]) {    // direction의 원소를 순서대로 본다

        case UP:    // 위 방향으로
            if (row - 2 >= 0 && !visited[row - 2][col]) {   // 접근 가능한지 판단하고
                maze[row - 1][col] = ' ';   // 벽을 없앤 후
                total_wall--;   // 벽의 총 개수를 1 감소하고
                createMaze(row - 2, col);   // 재귀적으로 위쪽 방에 대해 다시 한번 이 과정을 수행한다
            }
            break;

        case DOWN:  // 아래 방향으로
            if (row + 2 < maze_row && !visited[row + 2][col]) { // 접근 가능한지 판단하고
                maze[row + 1][col] = ' ';   // 벽을 없앤 후
                total_wall--;   // 벽의 총 개수를 1 감소하고
                createMaze(row + 2, col);   // 재귀적으로 아래쪽 방에 대해 다시 한번 이 과정을 수행한다
            }
            break;

        case LEFT:  // 왼쪽 방향으로
            if (col - 2 >= 0 && !visited[row][col - 2]) {   // 접근 가능한지 판단하고
                maze[row][col - 1] = ' ';   // 벽을 없앤 후
                total_wall--;   // 벽의 총 개수를 1 감소하고
                createMaze(row, col - 2);   // 재귀적으로 왼쪽 방에 대해 다시 한번 이 과정을 수행한다
            }
            break;

        case RIGHT: // 오른쪽 방향으로
            if (col + 2 < maze_col && !visited[row][col + 2]) { // 접근 가능한지 판단하고
                maze[row][col + 1] = ' ';   // 벽을 없앤 후
                total_wall--;   // 벽의 총 개수를 1 감소하고
                createMaze(row, col + 2);   // 재귀적으로 오른쪽 방에 대해 다시 한번 이 과정을 수행한다
            }
            break;
        }

    }
}
```

ofApp::createMaze()는 recursive backtracking algorithm 으로 완전 미로를 구성합니다. 이 함수의 알고리즘은 다음과 같습니다.

1. visited[row][col]에 true 를 저장함으로써 이번 노드가 방문되었음을 나타냅니다.
2. 상, 하, 좌, 우 방향으로의 재귀 호출 가능 여부를 판단하는 순서를 나타내는 배열 direction 을 정의하고 그 순서를 랜덤하게 섞습니다.
3. direction 의 원소를 순서대로 검토하며 해당 방향의 방이 아직 방문되지 않았는지 판단하고 벽을 없앤 후 total_wall 을 1 감소하고 재귀적으로 호출합니다.
4. 어떤 방향으로도 방문할 수 없다면 반환합니다.

이 알고리즘을 수행하면 랜덤한 완전 미로가 구성됩니다. 미로를 그래프라고 생각해 봅시다. 그러면 각각의 방은 노드이고, 상, 하, 좌, 우 각각의 방향으로 인접한 노드가 있을 수 있습니다. 따라서 이 알고리즘은 모든 방의 개수 M 에 대해 O(M)의 시간 복잡도와 공간 복잡도를 갖게 되는데, 방의 개수는 행의 개수 N 에 대해 N^2 와 비례하므로 시간 복잡도와 공간 복잡도는 앞선 기준에 따르면 O(N^2)라고 할 수 있습니다.

```
476  void ofApp::deinitializeMaze() {
477      maze.clear();   // maze를 초기 상태(빈 벡터)로 되돌리기
478      visited.clear();    // 방문 여부를 나타내는 벡터를 초기 상태(빈 벡터)로 되돌리기
479      HEIGHT = 0;     // 미로와 관련된 변수 0(NULL)으로 설정
480      WIDTH = 0;      // 미로와 관련된 변수 0(NULL)으로 설정
481      maze_row = 0;   // 미로와 관련된 변수 0(NULL)으로 설정
482      maze_col = 0;   // 미로와 관련된 변수 0(NULL)으로 설정
483      total_wall = 0;     // 미로와 관련된 변수 0(NULL)으로 설정
484  }
```

```
529  void ofApp::initializeGame() {
530      available_power = 0;    // 벽을 부술 수 있는 횟수 0으로 초기화
531      total_energy = 0;       // 현재까지 획득한 에너지 개수 0으로 초기화
532      minerX = minerY = 1;    // 캐릭터의 초기 위치를 (1, 1)로 고정
533
534      do {    // 미로의 방 위치에 있고 캐릭터의 위치와 겹치지 않는 좌표 생성
535          itemY = rand() % maze_row;  // 아이템의 행 위치 생성
536          itemX = rand() % maze_col;  // 아이템의 열 위치 생성
537      } while (itemY % 2 == 0 ||  // 2로 나누어 떨어지는 위치엔 벽이 있음
538               itemX % 2 == 0 ||  // 2로 나누어 떨어지는 위치엔 벽이 있음
539              (itemY == minerY && itemX == minerX));  // 캐릭터와 완전히 위치가 겹칠 수 없음
540  }
541
542  void ofApp::deinitializeGame() {
543      available_power = 0;    // 벽을 부술 수 있는 횟수 0으로 초기화
544      total_energy = 0;       // 현재까지 획득한 에너지 개수 0으로 초기화
545      minerX = minerY = 1;    // 캐릭터의 초기 위치로 초기화
546      easy_mode = hard_mode = false;  // 게임 모드 초기화
547  }
```

마지막으로 ofApp::deinitializeMaze(), ofApp::initializeGame(), ofApp::deinitializeGame()은 각각 미로와 관련된 정보를 초기 상태로 되돌리고, 게임과 관련된 정보를 초기 상태로 설정하며, 게임과 관련된 정보를 초기 상태로 되돌리는 함수입니다. 이 함수들의 동작은 주석이 더 자세히 설명하고 있으므로 생략하겠습니다.

4. 창의적 구현 항목

저는 이번 프로젝트를 진행함에 있어 수업 때 진행한 미로 프로젝트를 변형하여 미로를 게임판으로 만들었습니다. 이제까지 수업에서 배운 ofApp 클래스의 기본 함수의 기능을 모두 활용하였다고 보아도 무방할 것 같습니다. 단순히 미로에서 경로를 찾고 그 경로를 그리는 프로젝트를 하나의 게임으로 바꾼 것이 가장 창의적인 구현 항목이라고 생각합니다.

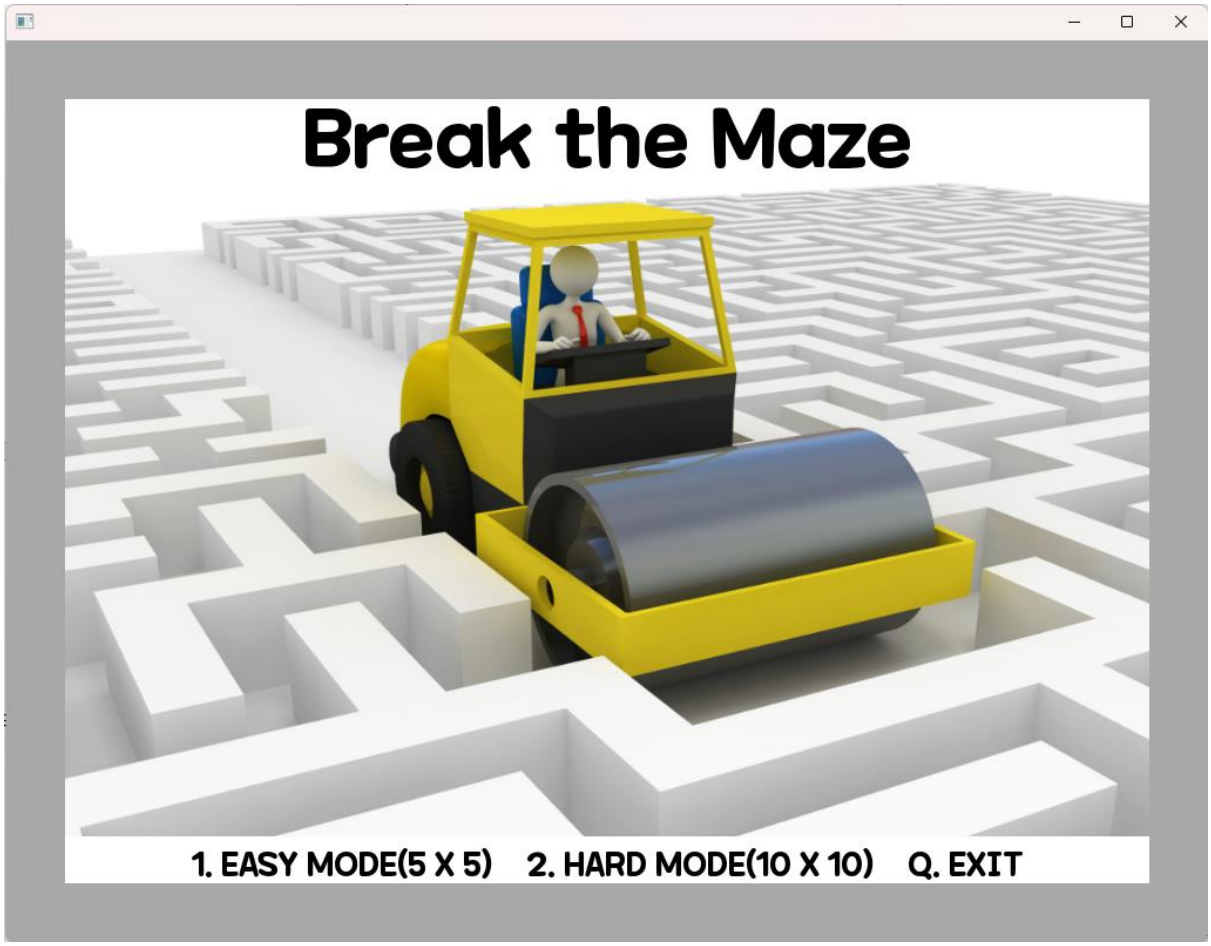
특히 캐릭터가 움직일 때 단순히 좌표를 한 번에 바꾸어 순간이동하는 것처럼 그려지지 않고 현재 가진 power에 비례한 속도로 미끄러지듯이 움직이는 sliding 알고리즘을 최소한의 변수만을 사용하여 구현한 것이 가장 독창적인 부분이었다고 생각합니다. 이는 실제로 복잡한 자료구조의 사용도 고려해 보았으나 조건문 몇 줄과 flag 변수 추가만으로 간단하게 구현할 수 있었습니다.

또한 미로의 크기가 작기 때문에 미로의 생성 자체는 과제 수행 시 선택하였던 알고리즘을 그대로 사용하여도 시간 복잡도와 공간 복잡도에 있어 차이가 생기지 않아 기존의 알고리즘을 차용했지만, 미로의 정보를 이용하여 게임 내의 객체를 생성하고 그 위치를 이동하는 등의 동작은 모두 가능한 적은 변수만을 사용하여 간단하게 구현하였습니다.

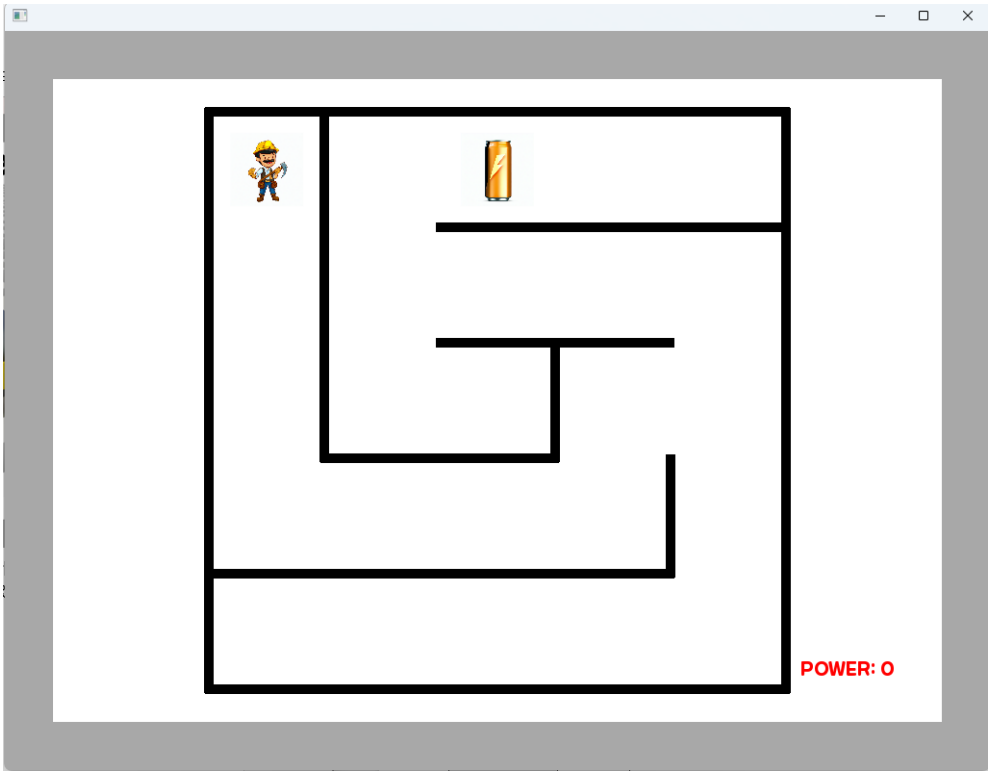
실제로 코드에서 미로를 생성하거나 그릴 때를 제외한 코드는 모두 O(1)의 시간 복잡도와 공간 복잡도를 갖습니다. 이는 가능한 연산을 적게 하기 위해 복잡한 자료구조를 사용하지 않고 웬만한 동작은 모두 간단한 변수를 최대한 활용하여 구현했기 때문입니다. 게임 구현에는 생각보다 복잡한 자료구조나 알고리즘은 필요하지 않았습니다.

5. 프로젝트 실행 결과

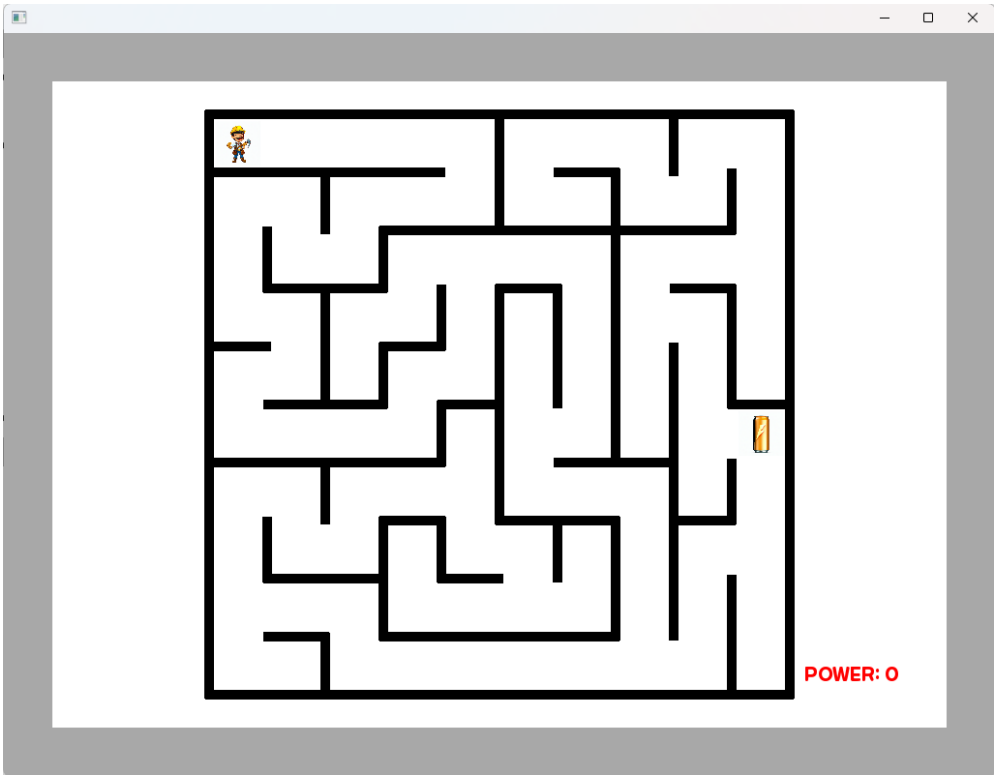
지금부터는 실제 게임 화면을 통해 게임이 어떻게 구현되었는지 검토해 보겠습니다.



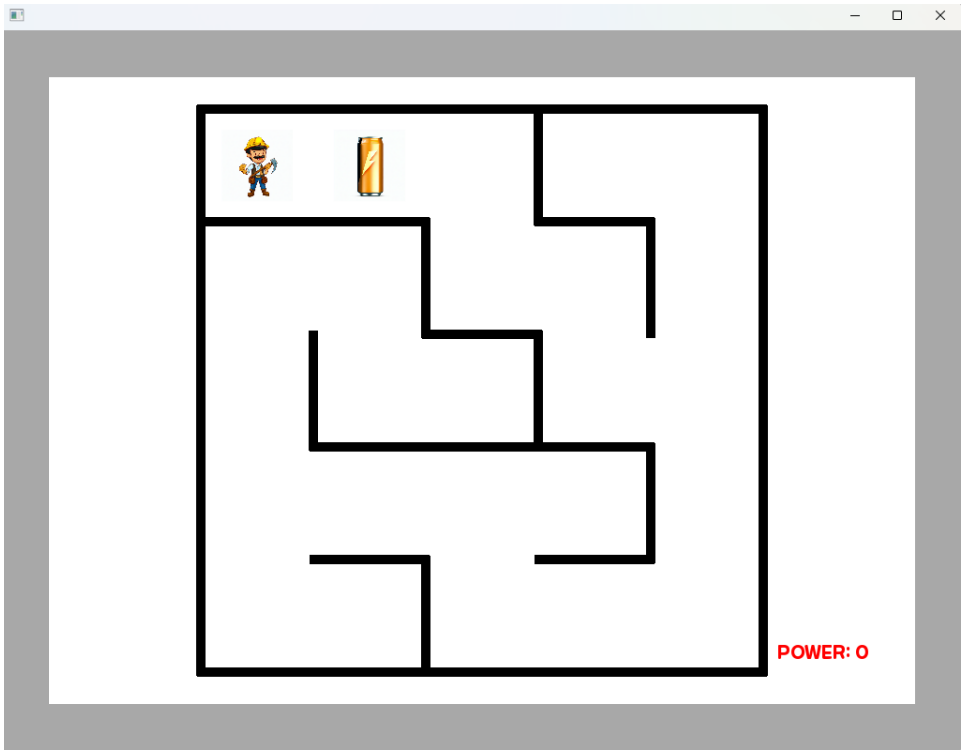
게임의 메인 화면입니다. 1 을 입력하면 이지 모드로 게임을 시작하고, 2 를 입력하면 하드 모드로 게임을 시작합니다. q 또는 Q 를 누르면 게임을 종료할 수 있습니다.



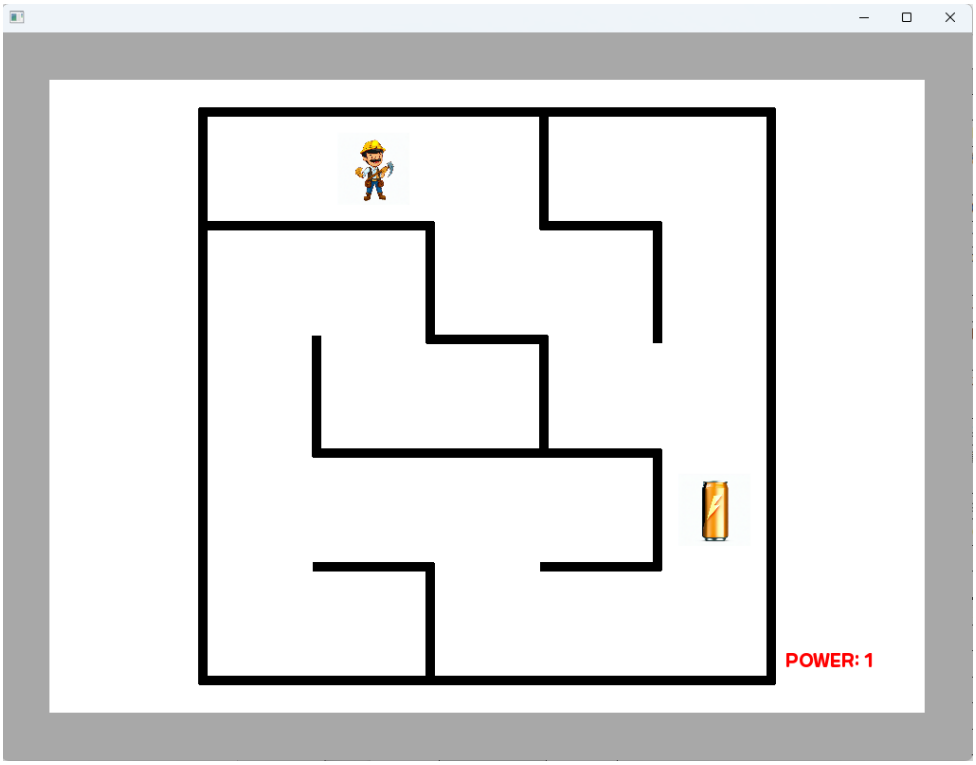
이지 모드 게임 플레이 화면입니다.



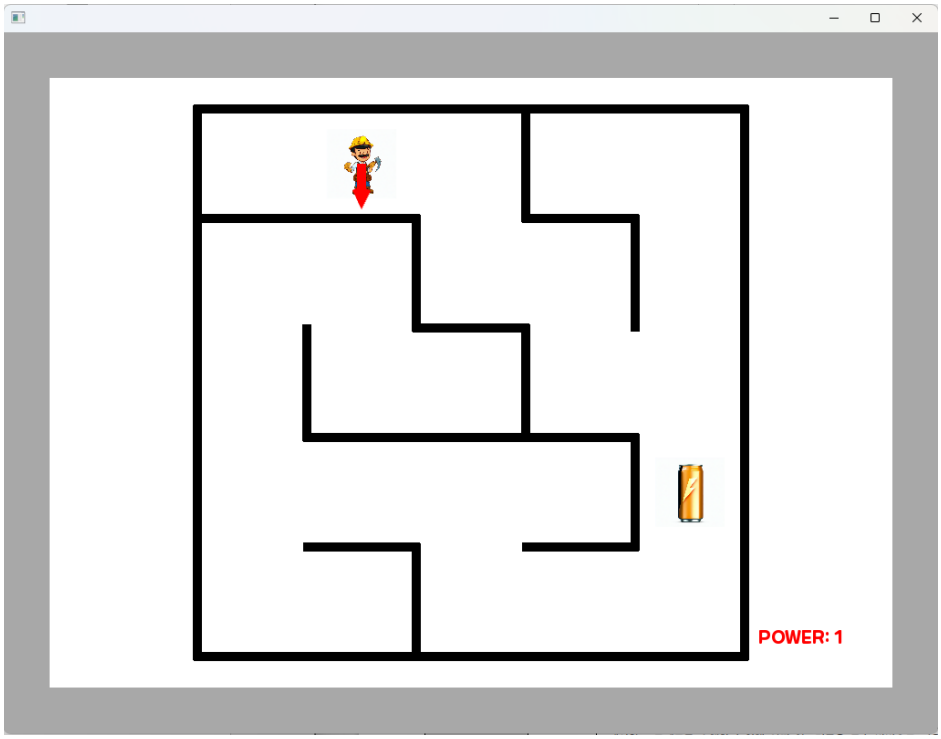
하드 모드 게임 플레이 화면입니다. 둘 다 플레이 방식은 같으므로 시각적인 변화를 확인하기 쉬운 이지 모드를 플레이하며 각각의 요소에 대해 설명해 보겠습니다.



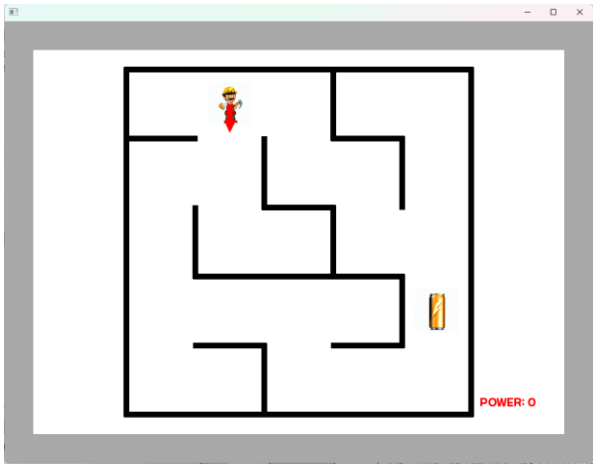
다시 이지 모드 플레이 화면입니다. 미로가 이전과 다른 형태로 다시 랜덤하게 생성되었고, 아이템의 위치도 역시 랜덤하게 결정되고 있습니다.

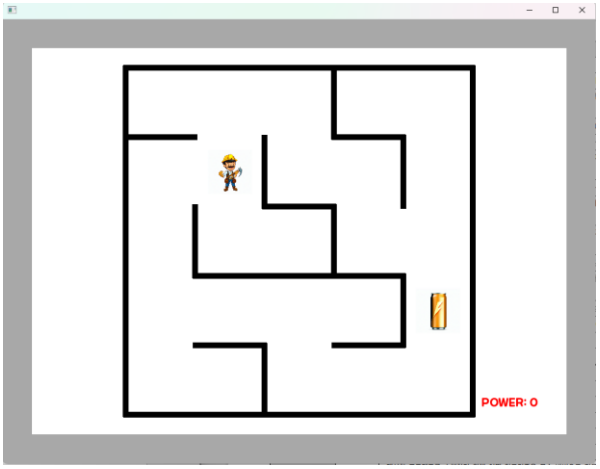


캐릭터가 아이템을 획득했을 경우의 화면입니다. POWER 가 1 올라갔고, 아이템은 다시 랜덤한 위치에 생성되었습니다. 캐릭터가 sliding 을 수행하는 것은 발표에서 보여드리도록 하겠습니다.

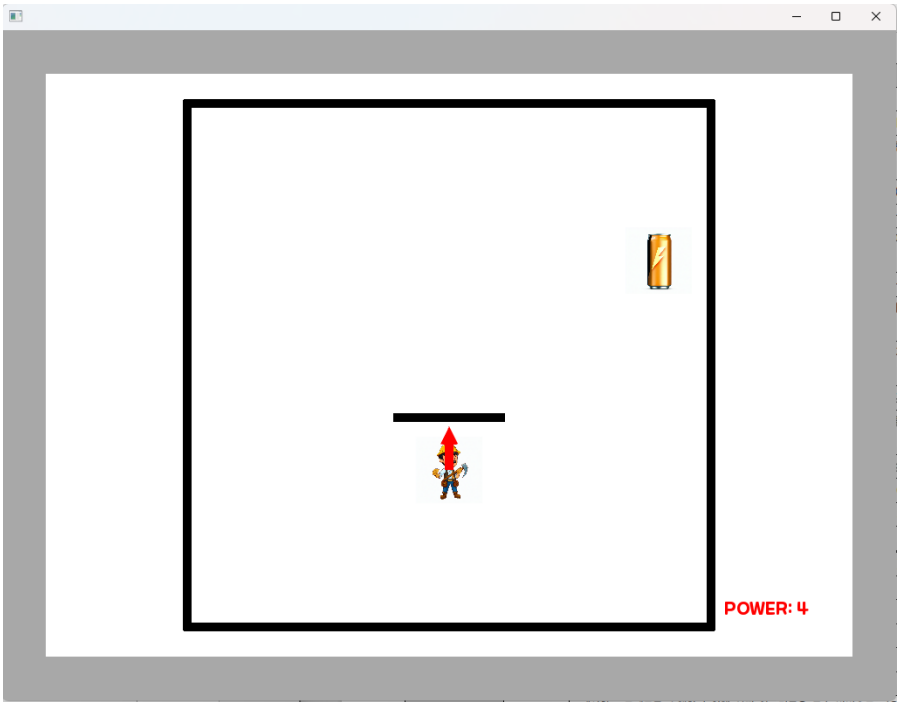


방향키를 누른 채로 떼지 않고 있으면 해당 방향으로 화살표가 그려집니다.

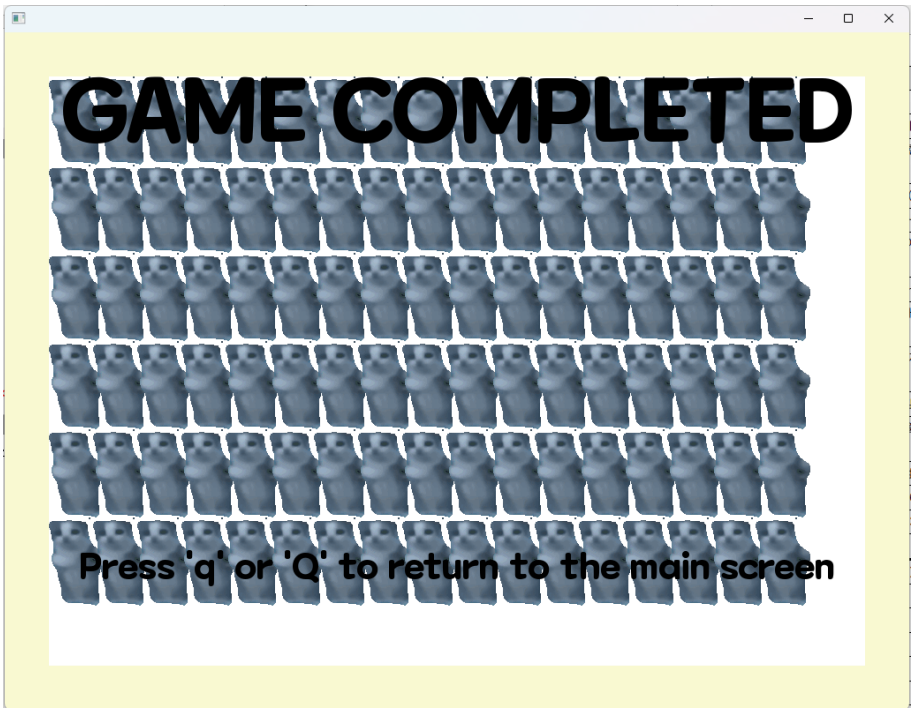




그 상태에서 스페이스 바를 누르면, POWER 가 1 이상일 때 POWER 를 하나 소모하면서 벽이 부숴지고 캐릭터는 부숴진 벽을 넘어 이동합니다. 사진으로는 확인할 수 없지만 POWER 에 비례하여 sliding 속도가 높아지기 때문에 벽을 부술수록 캐릭터는 점점 느려집니다.



이렇게 계속 벽을 부수다가 마지막 벽을 부수면 게임이 종료됩니다.



엔딩 화면입니다. 게임 플레이 중 q 또는 Q 를 입력하여 종료하면 엔딩 화면이 나오지 않고 메인 화면으로 이동합니다. 이 화면은 오직 게임을 플레이하여 모든 벽을 부숴야만 확인할 수 있습니다.

6. 느낀 점 및 개선 가능한 사항

본 프로젝트를 진행하면서 느낀 점과 추후에 조금 더 개선해볼 수 있는 점에 대해 서술해 보겠습니다.

첫 번째로 느낀 점은, 그래픽 구현에 있어 시간 복잡도와 공간 복잡도가 굉장히 중요한 고려 요소라는 것입니다. 일반적으로 온라인 게임을 플레이하려면 컴퓨터의 하드웨어를 최소 사양 정도는 맞추어야 끊김 없이 플레이할 수 있습니다. 제가 구현한 그래픽은 매우 간단한 수준이라 하드웨어에 과부하를 주진 않지만, 그래픽 요소가 복잡하고 세밀하게 구현된 게임들에선 한 프레임 출력 과정이 하드웨어에 엄청난 부하를 줄 것입니다. 또한 같은 맥락에서 모니터 주사율이 60Hz 인 경우보다 144Hz 인 경우에 하드웨어 자원을 더 소모하는 이유를 알게 되었습니다. 60Hz 는 1 초에 60 프레임만 그리면 되지만 144Hz 는 144 프레임을 그려야 하기 때문에 하드웨어가 그만큼의 연산을 뒷받침해 주어야 하는 것이었습니다.

두 번째로 느낀 점은, 프로젝트 개발에는 독창적인 아이디어를 생성해내는 것이 복잡한 알고리즘을 사용하는 것보다 중요한 역량이라는 것입니다. 제가 만든 게임은 내부적인 구현은 그렇게 복잡하지 않지만 분명히 저의 독창적인 아이디어를 빠짐 없이 반영하고 있습니다. 알고리즘이야 처음엔 비효율적인 것을 골랐더라도 나중에 개선하면 그만이기 때문에 이에 그렇게 큰 비중을 두는 것은 바람직하진 않은 것 같습니다.

프로젝트에서 개선할 만한 사항은 미로 알고리즘을 변경하는 것, 게임의 객체들(미로, 캐릭터, 아이템 등)을 객체 지향 프로그래밍 방식에 맞추어 구현하는 것을 생각해볼 수 있었습니다. 현재는 미로 알고리즘이 게임 시작 시 한 번만 수행되기 때문에 그 시간 복잡도에 대해 심각하게 고려할 필요는 없으나, 추후 미로를 여러 개 만든다거나 더 큰 미로를 만들어야 하는 상황이 오면 이는 게임 실행 속도에 영향을 줄 수 있습니다. 따라서 그 경우를 대비하여 현재와 같은 재귀 호출 알고리즘이 아닌 반복적인 알고리즘을 사용하는 방식으로의 개선이 필요할 수 있겠습니다. 그리고 같은 맥락에서 게임의 요소와 기능들이 많아질수록 절차지향적인 프로그래밍 방식으로 모든 걸 구현하기엔 한계가 있습니다. 따라서 프로젝트를 확장할 예정이라면 작은 규모일 때 미리 객체 지향 프로그래밍 방식으로 다시 구현하여 두는 것이 추후에 유지보수하기 좋을 것입니다.