1. 실험시간에 작성한 프로그램에서 자료구조와 구성한 자료구조를 화면에 그리는 방법들을 설명한다. 완성한 자료구조를 이용한 그래픽 전환 작업의 시간 및 공간복잡도를 보이고 실험 전에 생각한 방법과 어떻게 다른지 아울러 기술한다.

아래와 같이 char 타입의 2차원 vector를 생성해, .maz 파일로부터 그대로 입력받아 저장했다.

```
vector<vector<char>> maze;
```

FILE 버튼을 누르면 사용자가 입력 파일을 선택할 수 있다. readFile() 함수에서는 다음과 같이 파일을 한 라인씩 읽어들이면서 maze에 저장한다. 2차원 vector에 모두 저장한 뒤, ('+'까지 포함된) 이미지의 가로 길이(imageWIDTH), 세로 길이(imageHEIGHT)를 저장했다. 그리고 (imageWIDTH-1)/2, (imageHEIGHT-1)/2로 미로의 실제 가로 길이(mazeWIDTH), 세로 길이 (mazeHEIGHT)를 저장했다. 이러한 작업의 시간복잡도는 O(imageHEIGHT * imageWIDTH)이다.

```
/* Read file line by line. */
	string line;
	string::iterator iter;
	int i = 0;
	int j;

	/* Save maze */
	while (getline(input_file, line)) {
	   maze.push_back(vector<char>());
	   for (iter = line.begin(); iter != line.end(); iter++) {
	      maze[i].push_back(*iter);
	   }
	   i++;
	}

	/* Save WIDTH */
	imageWIDTH = maze[0].size();

	/* Save HEIGHT */
	imageHEIGHT = i;

	/* Close */
	input_file.close();

	/* Save maze size */
	mazeWIDTH = (imageWIDTH - 1) / 2;
	mazeHEIGHT = (imageHEIGHT - 1) / 2;
```

그리고 읽어들인 정보를 바탕으로 화면에 미로를 선으로 그려줄 때 사용할 좌표를 용이하게 저장하기 위해 MazeWall 구조체를 선언하고, MazeWall 타입의 vector를 생성했다.

```
struct MazeWall{
   float startX;
   float startY;
   float endX;
   float endY;
};
```

```
vector<MazeWall> walls;
```

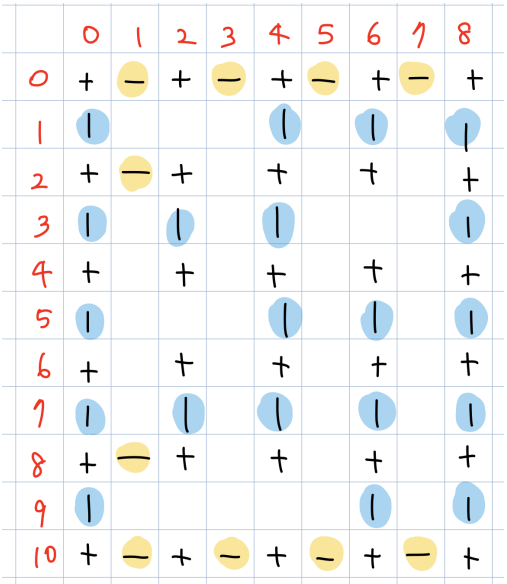
readFile() 함수에서 저장한 미로 정보를 바탕으로 2차원 vector를 순회하며 '|' 또는 '-'를 찾은 경우 세로 선의 시작점 및 끝점의 좌표를 저장했다. 그리고 true를 반환했다. 이러한 작업의 시간복잡도는 O(imageHEIGHT * imageWIDTH)이다.

```
/* Save coordinate information */
	MazeWall tmp;

	for (i = 0; i < imageHEIGHT; i++) {
		for (j = 0; j < imageWIDTH; j++) {
			switch (maze[i][j])
			{
				case '|':
					tmp.startX = j + 0.5;
					tmp.startY = i - 0.5;
					tmp.endX = j + 0.5;
					tmp.endY = i + 1.5;
					walls.push_back(tmp);
					break;
				case '-':
					tmp.startX = j - 0.5;
					tmp.startY = i + 0.5;
					tmp.endX = j + 1.5;
					tmp.endY = i + 0.5;
					walls.push_back(tmp);
					break;
				default:
					;
			}
		}
	}

	return true;
```

위와 같이 저장할 수 있는 이유는 다음과 같이 '-', '|'를 발견한 경우 그 인덱스를 활용해 위치 정보를 저장할 수 있기 때문이.

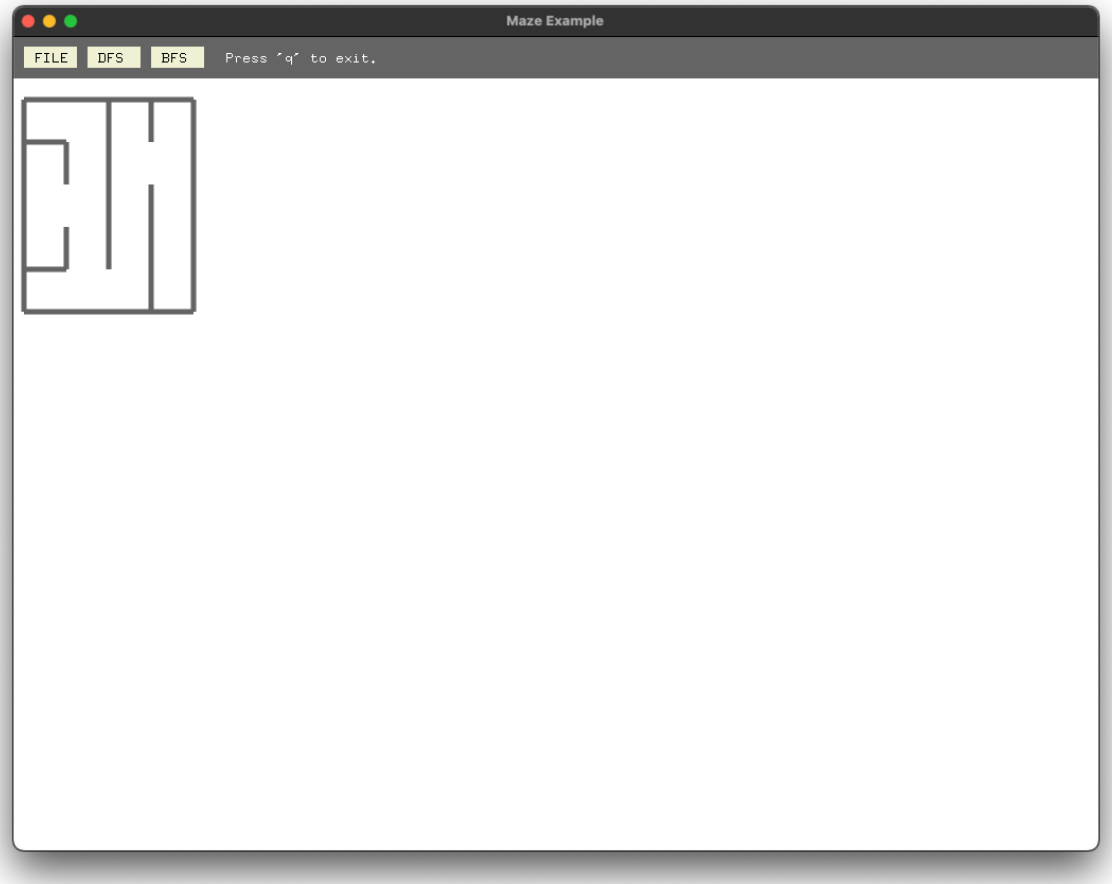


성공적으로 파일을 읽어들였다면 isOpen이라는 flag 변수가 1이 된다.

draw() 함수에서는 flag 변수가 1인 경우, 저장한 벽의 양 끝 좌표를 바탕으로 선을 그려주었다. 좌표에 20을 곱해서 벽의 길이를 늘렸고, y좌표는 50씩 더해서 화면 위쪽의 메뉴 공간보다 아래에 그려지도록 했다. 이러한 작업의 시간복잡도는 O(mazeHEIGHT * mazeWIDTH)이다.

```
/* Draw maze. */
    if (isOpen) {
        ofSetColor(100);
        for(int i = 0; i < walls.size(); i++) {
            ofDrawLine(20 * walls[i].startX, 50 + 20 * walls[i].startY, 20 * walls[i].endX,
50 + 20 * walls[i].endY);
        }
    }
```

다음과 같이 미로를 화면에 나타낼 수 있었다.



즉 미로를 파일로부터 입력받아 화면에 그리는 작업의 시간복잡도는 O(imageHEIGHT * imageWIDTH)인데, imageHEIGHT는 O(mazeHEIGHT)이고 imageWIDTH는 O(mazeWIDTH)이므로, 실제 미로의 너비를 WIDTH, 높이를 HEIGHT라고 한다면 시간복잡도는 O(WIDTH * HEIGHT)이다.

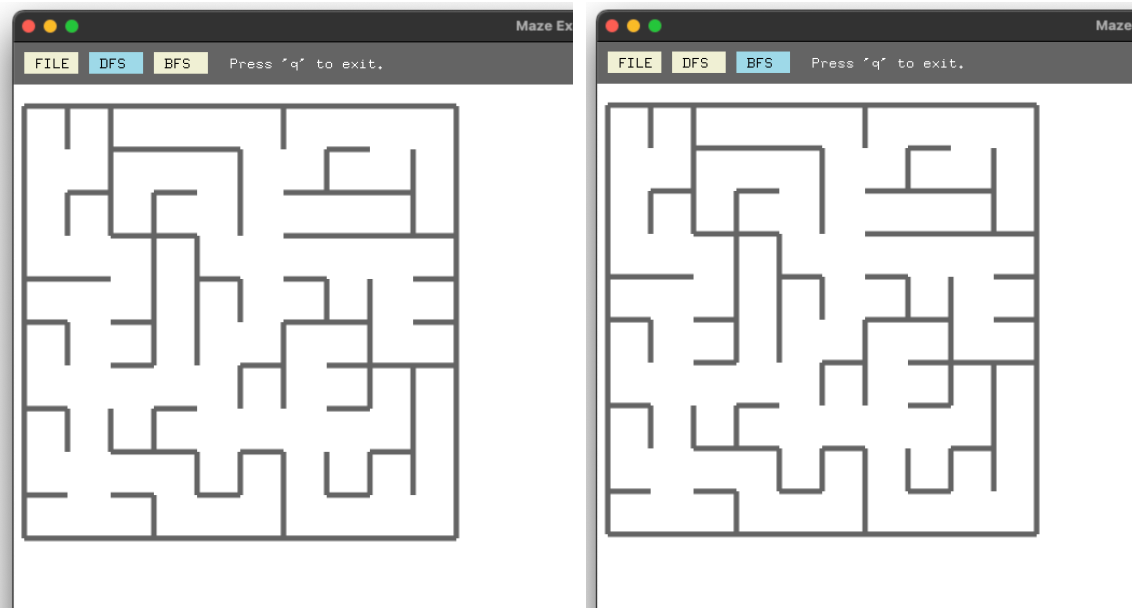
그리고 전체 이미지를 입력받는 과정의 공간복잡도는 O(imageHEIGHT * imageWIDTH), 미로의 벽을 저장하는 과정의 공간복잡도는 O(mazeWIDTH * mazeHEIGHT)이므로 마찬가지로 미로를 파일로부터 입력받아 화면에 그리는 작업의 공간복잡도는 O(WIDTH * HEIGHT)이다.

전체적인 알고리즘은 예비보고서에서 작성한 pseudo code와 동일했다.

2. 본 실험 및 숙제를 통해 습득한 내용을 기술하시오.

Maze를 생성하는 프로그램에서 문자로 출력물을 만드는 것과 달리, openFrameworks를 사용할 때는 도형을 통해 화면에 미로를 그려주어야 했다. 단순히 특정 문자를 출력하는 것이 아니므로 어떻게 화면에 미로를 그려줄 수 있을지 고민하는 시간을 가질 수 있었으며, 좌표를 저장해서 그려주는 방법이 있다는 것을 알 수 있었다.

한편 3주차에서 DFS, BFS를 사용하게 되는데, 이러한 기능을 켜고 끄는 메뉴 혹은 버튼이 필요해서 이번 주차에서 구현하게 되었다. Waterfall 실험에서 구현한 대로 keyPressed 방식을 사용해 DFS 모드, BFS 모드를 전환하는 방법이 있지만, 새롭게 메뉴 형태를 띠는 버튼을 구현하는 방식을 알아보았다. 예비보고서에 작성한 대로 ofxGui addon을 활용해 버튼 혹은 메뉴를 만드는 것이 가능하다. 하지만 이 방법은 불필요한 헤더 파일을 많이 포함하게 되어 최대한 간단하게 버튼을 구현하는 방법을 알아보게 되었다. Waterfall 실험에서 화면 위에 사각형으로 ceiling을 만들었던 방식으로 menu bar를 만들고, 직사각형 3개를 만들어서 button처럼 작동하게 만들었다. 즉 mousePressed() 함수에서 inside() 함수를 활용해, 해당 사각형(버튼) 안에서 마우스를 클릭하면 특정한 코드를 작동시킨다면 버튼처럼 사용할 수 있다. isDFS, isBFS라는 flag 변수를 선언하여 둘 다 켜지는 일은 막고, DFS 버튼을 누른 경우 DFS 모드, BFS 버튼을 누른 경우 BFS 모드로 들어갈 수 있도록 코드를 작성했다.



즉 keyPressed 방식 외에도 입력받을 수 있는 장치를 만드는 방법을 습득할 수 있었다.