2022 1학기 컴퓨터 공학 설계 및 실험 1 최종 프로젝트

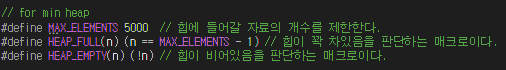
전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181288 이름: 윤성호

1. <프로젝트 목표, 실험 환경에 대한 설명>

컴퓨터 공학 설계 및 실험 1 최종 프로젝트로는 후반에 했던 maze project를 이용하여 실제 플레이가 가능한 미로 탈출 게임을 제작하였다. 이 프로젝트를 통해, 오픈프레임워크를 이용하여 필요한 것을 화면에 그리고, 또한, 게임을 진행할 때, 수업 시간에 사용했던 알고리즘 이외의 알고리즘을 c언어를 사용하여 구현하면서, 프로그래밍 실력을 키워보고자 하였다. 엔터를 눌러 게임을 시작하고, 매 스테이지마다 빨간점이 4개가 생성되는데, 이를 모두 먹으면 다음 스테이지로 넘어간다. 다음 스테이지로 넘어갈 때는 게임의 난이도를 점점 상승시키기 위해 미로의 가로세로 사이즈를 1씩 증가시킨다. 즉, 스테이지 1에서는 4X4 미로, 스테이지 2에서는 5X5 미로가 엘러 알고리즘에 의해 무작위로 생성되어 플레이어에게 제공된다. 또한, 스테이지가 올라갈수록 게임의 난이도가 기하급수적으로 증가하기 때문에 플레이어의 의욕을 잃지 않게 하기 위해 “힌트” 를 제공하는데, 이는 현재 플레이어의 위치에서 가장 짧은 경로로 남은 빨간점들을 모두 먹을 수 있는 경로를 2초간 보여주고 서서히 경로를 사라지게 했다. 즉, 이번 프로젝트에서는 방향키로 플레이어가 대상을 이동시키는 경험을 할 수 있을 뿐 아니라, 스테이지 별 게임 난이도가 올라감에 따라, 혼자의 힘으로 깨는 것이 힘들 경우를 대비해 미로 탈출 경로를 힙과 disjoint set을 결합한 크루스칼 알고리즘을 이용하여 최소 신장 트리를 생성해 정답 경로를 제공하였다. 프로그램 구현에 대한 자세한 설명은 5번에서 하도록 하겠다. 이번 컴실 1학기 최종 프로젝트는 visual studio 2017 상에서 실행되도록 구현하였다.

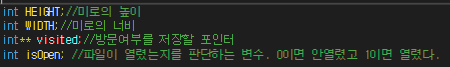
1. <각 변수에 대한 설명>

프로그램을 실행시키는 데 필요한 전역 변수들의 선언은 모두 ofApp.h에 미리 해놓았다. 선언만 해놓았을 뿐, 실제 값을 대입하거나 내용을 추가하는 작업은 ofApp.cpp에서 진행하려고 한다. 프로그램을 작성할 때 헤더파일에는 변수와 함수의 선언만이 오는 것이 오는 것이 좋다는 것을 22-봄학기 컴퓨터공학 실험 및 설계 과목에서 배웠고, 이렇게 하는 것이 코드가 더 깔끔해진다는 것을 실습을 진행하며 몸소 느꼈기 때문에 그러한 과정으로 프로그램을 작성하였다.

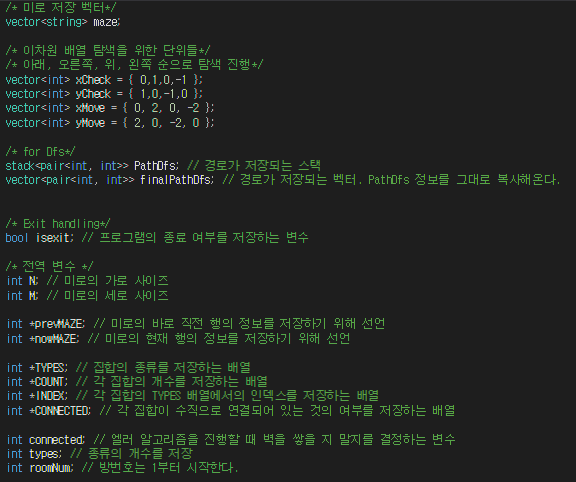


[사진 1]

최종 프로젝트에서는 게임의 난이도가 올라갈 경우를 대비하여 플레이어의 현재 위치에서 빨간점을 모두 먹을 수 있는 최적의 경로를 추천해준다. 이를 구현하기 위해 크루스칼 알고리즘을 이용하여 최소 신장 트리를 생성하였다. 크루스칼 알고리즘의 효율성을 고려하기 위해 힙 자료구조를 활용하였고, [사진 1]에는 이를 위해서 필요한 변수들을 ofApp.h에서 정의한 것을 보여주는 캡처본이다. MAX\_ELEMENTS는 힙에 들어갈 자료의 개수를 제한해주기 위해 만들었다. 5000개로 제한하였다. 게임에 대한 숙련도가 높은 플레이어가 등장하면 크기가 아주 큰 힙이 필요할 수 있지만, 그러나, 시간제한을 두어 그렇게 큰 미로가 생성되는 것은 불가능하도록 게임을 제작하였다. HEAP\_FULL 은 힙이 꽉 차있음을 판단하는 매크로로, 힙 자료구조에 노드를 추가할 때 힙이 꽉 차있으면 더 이상 자료를 추가할 수 없기 때문에 그러한 에러 처리를 해주기 위해 만든 매크로이다. 같은 느낌으로 HEAP\_EMPTY는 힙이 비어있음을 판단하는 매크로로, 힙에서 노드를 삭제할 때 힙이 비어있다면 자료를 삭제할 수 없기 때문에 그러한 에러 처리를 해주기 위해 만든 매크로이다.

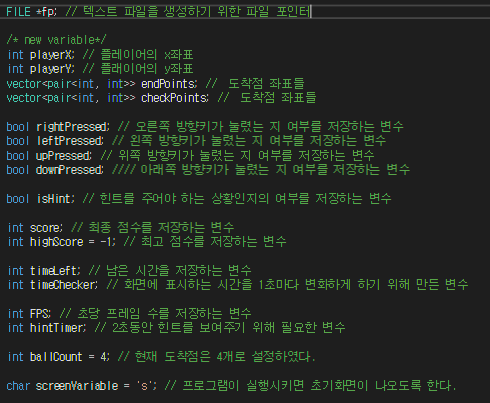


[사진 2]



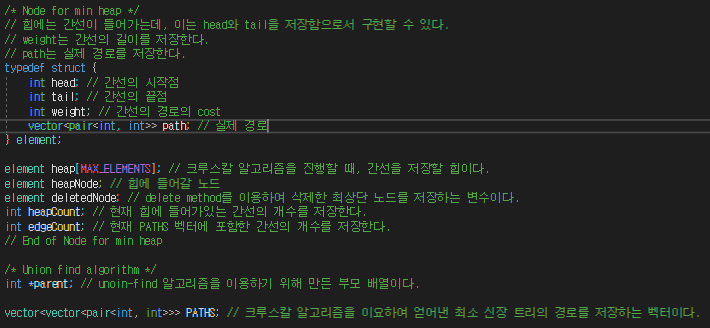
[사진 3]

[사진 2]와 [사진 3]은 엘러 알고리즘을 이용하여 미로를 생성할 때 필요한 변수들을 ofApp.h에 정의한 것을 캡처한 것이다. 가장 먼저, HEIGHT 와 WIDTH는 각각 새롭게 생성되는 미로의 높이와 미로의 너비를 저장한다. 그리고, visited 배열은 방문여부를 저장하는 배열인데, 이는 크루스칼 알고리즘을 진행하기 위한 전처리 과정을 진행하는 dfs 과정에서 필요한 배열이다. 각 지점사이의 경로와, 경로의 크기를 얻어오기 위해 필요한 배열이다. 다음으로는 isOpen변수인데, 이는 파일이 열렸는지의 여부를 판단하는 변수이다. 0이면 열린 것이고, 1이면 닫힌 것이다. 다음으로 maze라는 이름의 string 자료형을 원소로 가지는 vector 자료구조이다. 이 벡터에는 미로를 저장하는데, .maz 파일로 받아오기 때문에 각 줄을 문자열의 형태로 받아오는 것이 편리하기 대문에 string 자료형을 원소로 가지도록 하였다. 미로를 매 스테이지 마다 새롭게 생성하는 만큼 maze의 사이즈는 계속해서 변화하게 된다. 이렇듯 변화하는 사이즈의 자료들을 저장하는데 c++에서는 vector 자료구조를 제공한다. c언어에서의 배열과 마찬가지로 각 원소에 인덱스로 접근이 가능하고, 사이즈 또한 유동적으로 변경이 가능한 유용한 자료구조이다. 따라서, 미로를 저장하는데 벡터 자료구조를 사용하였다. 그 다음, xCheck, yCheck 벡터는 미로의 그 다음 위치가 벽인지 아닌지를 체크한다. 미로에서는 상, 하, 좌, 우 총 4가지 방향으로 이동이 가능하다. 즉, 4가지 뱡향에 벽이 있는지 체크해야함을 의미하는데, 원래는 이를 4개의 if문을 이용하여 체크를 해야한다. 그러나, 체크 방향을 두개의 벡터로 저장해두고, 이 두개의 벡터를 순회하면서 상, 하, 좌, 우를 체크하면 코드가 훨씬 간략해지고 디버깅도 편리해지는 장점이 있다. xCheck와 yCheck를 탐색하며 벽의 유무를 체크하고 이동이 가능하면 xMove, yMove 배열을 이용하여 실제로 다음 위치로 이동한다. 체크하는 벡터와 이동시키는 벡터를 따로 둔 이유는 미로는 빈칸과 벽으로 이루어져 있기 때문에 빈칸과 빈칸 사이의 간격은 2이다. 따라서, 벽은 빈칸 사이의 간격을 1이고 빈칸과 빈칸 사이의 간격은 2이다. 따라서, xMove와 yMove는 2만큼 이동하도록 벡터의 구성요소들을 설정하였다. 다음으로 PathDfs인데 이는 dfs함수를 한번 돌리면 두 점사이의 이동 경로를 구할 수 있는데, 이를 저장하는 스택이다. 크루스칼 알고리즘의 전처리 과정에서 사용된다. 그런데 PathDfs는 스택 자료구조로 저장되어 있어 인덱스로 접근이 불가능하기 때문에 다루기가 쉽지 않아서 finalPathDfs라는 벡터 자료구조를 새롭게 선언하여 PathDfs의 모든 원소들을 옮긴다. Isexit은 프로그램이 종료되었는징 여부를 저장하는 변수이다. 이를 이용해 프로그램이 종료됨을 알 수 있다. 그리고 N, M은 .maz 파일에서 받아올 미로의 사이즈를 저장하는 변수이다. 나중에 init함수에서 초기화를 진행하는데 초기화는 4로 진행한다. preMaze는 미로의 바로 직전 행의 정보를 저장하기 위해 선언하였다. 또한, nowMAZE는 미로의 현재 행의 정보를 저장하기 위해 선언하였다. 이는 엘러 알고리즘을 이용하여 미로를 생성하는데 전체적인 미로를 저장할 필요는 없고, 바로 미로의 바로 인접한 행들만을 저장해도 미로를 구현하는데는 문제가 없다고 생각하였기 때문에 만들었다. 이를 통해 메모리를 아끼는 효과를 볼 수 있다. TYPES는 집합의 종류를 저장하는 배열로, 엘러 알고리즘의 세로 경로를 생성하기 위해 필요한 배열이다. 어떤 집합들이 저장되어 있는지 방번호 형태로 저장되게 된다. COUNT는 각 집합의 개수를 저장하는 배열로 이 또한 엘러 알고리즘의 세로 경로를 생성하기 위해 필요한 배열이다. 각 방 번호가 몇 개씩 있는지를 저장하는 배열이다. INDEX는 각 집합의 TYPES 배열에서의 인덱스를 저장하는 배열이다. TYPES와 COUNT 배열을 모두 파이썬의 딕셔너리 자료구조와 같이 사용하기 위한 해싱 과정을 이용한 것이다. INDEX 배열에 각 집합의 방번호가 저장되어 있는 인덱스를 저장하고 있어, TYPES와 COUNT 배열의 값에 O(1)의 시간복잡도만에 접근할 수 있다. CONNECTED는 각 집합이 수직으로 연결되어 있는 것의 여부를 저장하는 배열이다. 이를 통해 이미 연결한 방번호는 연결을 할 수도 있고 안할 수도 있지만, 아직 연결되어 있지 않는 방번호는 필수적으로 연결을 시켜줄 수 있다. connected는 정수형 변수로서 엘러 알고리즘을 진행할 때 벽을 쌓을 지 말지를 결정하는 변수이다. 이 변수는 가로 벽을 세울 대나 세로 벽을 세울 때 모두 이용하는 변수이다. types는 종류의 개수를 저장한다. 또한 roomNum은 방번호를 저장하는데 1부터 시작한다. 초기화과정은 실제 cpp파일에서 진행한다.



[사진 4]

[사진 4]는 미로 탈출 게임을 진행하는 데 필요한 변수들을 ofApp.h에 선언한 모습을 캡처한 것이다. 가장 먼저, 파일포인터 fp를 선언하였는데, .maz 파일을 생성하기 위한 파일 포인터이다. 이를 이용해 텍스트 파일 형태로 미로를 생성한다. 그리고 player, playerY는 각각 플레이어의 현재 x좌표, y좌표를 나타내고, 이는 플레이어가 이동함에 따라 매순간 변화하는 변수이다. endPoints는 최종 목표인 빨간점들의 위치를 담고 있는 벡터이다. 각각 x좌표와 y좌표를 담고 있어야 하기에 pair 자료구조를 이용하여 각각의 원소마다 정수형 변수를 두 개씩 담았다. 다음으로는 checkPoints라는 벡터인데, 이는 빨간점들의 위치 및 현재 플레이어의 위치를 모두 담는다. 그 이유는 checkPoints 벡터에 들어있는 위치들을 대상으로 하여 최소 신장 트리를 생성하여 힌트 경로를 추천해줄 것이기 때문이다. 이번 프로젝트에서 가장 중요한 변수라고 생각한다. 다음으로는 rightPressed, leftPressed, upPressed, downPressed이다. 이 변수들은 각각 오른쪽, 왼쪽, 위쪽, 아래쪽 방향키가 눌렸는지의 여부를 저장하는 변수이다. 누른 방향으로 이동을 해야하는데, 윈도우 특성상 한쪽 방향키를 꾹 누르고 있으면 실제 입력이 한번 된 이후, 약간의 시간 텀이 있는 다음에 연속적인 입력이 된다. 이러한 문제 때문에 플레이어가 게임을 진행할 때 방향키를 바꿔야 하는 상황이 빈번하게 나타나는데 이 때마다 방향키를 누른 방향으로 한칸 움직인 후 잠시 멈췄다가 쭉 움직이게 되는 상황이 생긴다면, 게임에 대한 피로도가 급격히 쌓일 것이다. 결과적으로 원하는 것은, 방향키를 눌렀을 때는 부드럽게 이동하고, 여러 방향키를 한번에 인식하게 하는 것(대각선으로 이동하기)도 구현하고 싶었기 때문에 이러한 변수를 추가로 두었다. 다음으로 isHint 변수는 힌트 경로를 제공하는지의 여부를 저장하는 변수로, 스페이스바를 누르면 isHint가 2초간 true로 바뀌고 평소에는 계속 false상태를 유지한다. 그리고 score은 최종 점수를 저장하는 변수이고, highScore은 최고 점수를 저장하는 변수이다. timeLeft는 현재 게임 플레이를 할 수 있는 남은 시간을 저장한다. 게임 플레이 시간은 init함수에서 30초로 초기화한다. FPS는 초당 프레임 수를 저장하는 변수이고, hintTimer는 2초동안 힌트를 화면에 보여주기 위해 필요한 변수이다. ballCount는 게임 상에 등장할 빨간점의 개수를 저장하는 변수이고, 이번 프로젝트에서는 4개의 빨간점을 가지고 게임을 진행해보려고 한다. screenVariable은 현재 어떠한 화면에 있는지를 나타내는 변수이다. ‘s’ 면 초기화면이고 ‘g’는 게임화면이다.

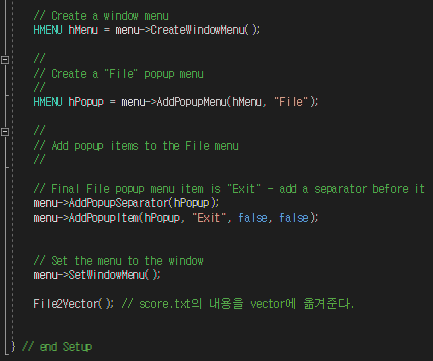
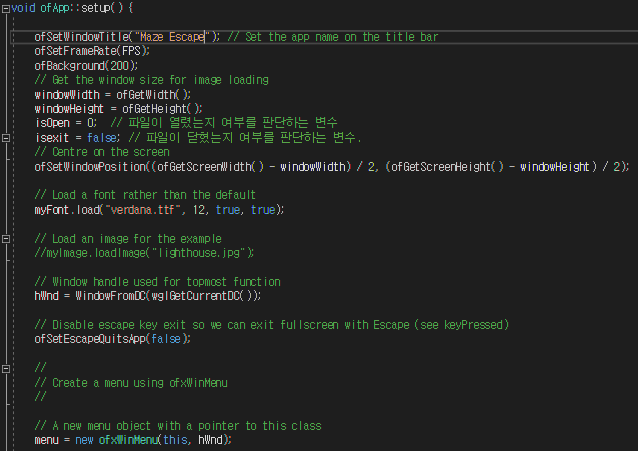


[사진 5]

이번 프로젝트에서는 크루스칼 알고리즘의 시간복잡도를 향상시키기 위해 힙 자료구조를 이용하였다. 힙 자료구조는 모든 노드의 기준 자료의 값이 자식노드의 기준 자료의 값보다 큰 형태를 max heap이라고 하고, 모든 노드의 기준 자료의 값이 자식노드의 기준 자료의 값보다 작은 형태를 min heap이라 한다. 현재 상황에서는 min heap을 구현하였다. 이를 위해서 힙에 들어가야 할 노드의 자료형을 정의해주어야 하고, 이는 element라는 이름의 자료형으로 사용할 수 있도록 구조체를 정의하였다. 구성 변수로는 간선의 시작점을 나타내는 정수형 변수 head, 간선의 끝점을 나타내는 정수형 변수 tail, 간선의 경로의 cost를 나타내는 정수형 변수 weight, 실제 경로를 나타내는 벡터 자료구조로 정의한 path가 있다. 그리고, 힙을 구성하는데, element 자료형이 힙에 들어가므로 element heap라고 정의하였다. 힙의 사이즈는 MAX\_ELEMENTS로 매크로에서 5000으로 정의하였다. 다음으로는 heapNode와 deletedNode이다. heapNode 힙에 들어갈 노드를 선언한 것이므로, heapNode를 이용하여 원소를 추가한다. 또한, deletedNode는 힙의 삭제 과정에서 삭제된 최상단 노드를 담는 변수이다. 그 다음으로 heapCount는 현재 힙에 들어가있는 간선의 개수를 저장하고, edgeCount는 크루스칼 알고리즘을 통해 얻어진 간선이 PATHS에 추가된 개수를 저장한다. 그리고, 크루스칼 알고리즘을 효율적으로 작성하기 위해서는 힙 자료구조뿐만 아니라 unoin-find 알고리즘또한 사용하였다. 크루스칼 알고리즘은 최소 비용의 간선부터 차례로 추가하는데, 이 때, 사이클이 발생하면 안되기 때문에 매번 체크해주어야 한다. 사이클 체크는 두 집합을 union 할때 parent값이 같으면 사이클이 발생한 것이다. 이를 위해 모든 노드의 부모가 되는 루트 노드를 저장할 필요가 있고, 이를 위해 parent라는 배열을 정의하였다. 마지막으로 PATHS 크루스칼 알고리즘을 이용하여 얻어낸 최소 신장 트리를 저장하는 벡터이다.

1. <각 함수에 대한 설명>

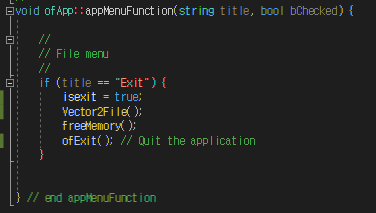
* setup 함수



[사진 6]

[사진 6]은 setup 함수를 구성하는 코드를 캡처한 사진이다. 오픈 프레임워크를 처음 실행했을 때 호출되는 함수이다. 프로그램의 초기화 과정이 들어가는 함수이다. 가장 먼저, Maze Escape 이라는 이름의 창을 띄우기 위해 ofSetWindowTitl e 매서드를 이용하였다. 그 다음 초당 프레임 수를 ofSetFrameRate 함수를 이용하여 설정해주었다. 초당 프레임 수는 FPS라는 변수안에서 설정해주며, 이는 init 함수에서 초당 50 프레임이 뜨도록 설정해주었다. 그리고 바탕색은 옅은 회색으로 칠해주기 위해 ofBackGround(200)을 해주었다. 이후, 윈도우 창의 가로 세로 길이를 얻어오기 위해 ofGetWidth, ofGetHeight 함수를 이용하였고, 미로 파일이 열려있는 지 여부를 저장하고 있는 isOpen 변수는 0으로 초기화한다. 프로그램을 시작하자마자는 미로 파일이 열려있지 않기 때문이다. 그리고 isexit 이라는 변수를 두어 프로그램의 종료 여부 또한 저장할 수 있도록 했다. 그리고, 화면을 중앙에 뜨도록 맞춘 다음, 글씨를 쓸 폰트는 verdana.tff로 설정해주었다. 이 후, 메뉴바 또한 생성하였는데, 기존의 maze project에서 쓰던 대부분의 메뉴를 삭제하고, exit 메뉴 하나만 남겨두었다. 그리고 가장 마지막 부분에 File2Vector 함수를 호출했는데, 이 함수는 오픈프레임워크 폴더안에 score.txt 파일에 이전 게임의 기록이 저장되어 있다면 프로그램 내부 메모리로 불러오는 과정이다. 만약, score.txt 파일이 없다면, 아무것도 불러오지 않는다. 이 함수에 대해서는 뒤에서 다시 설명하도록 하겠다.

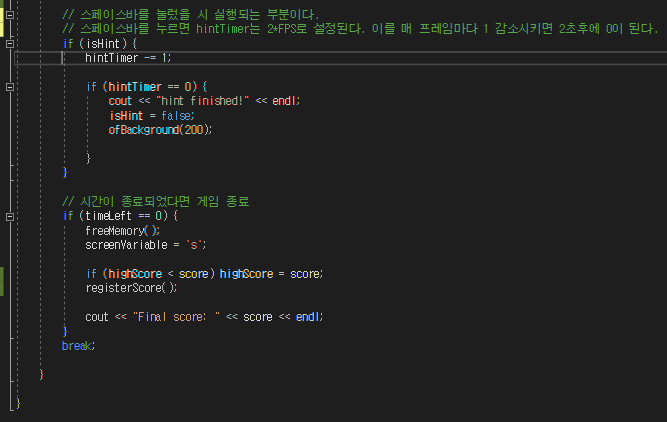
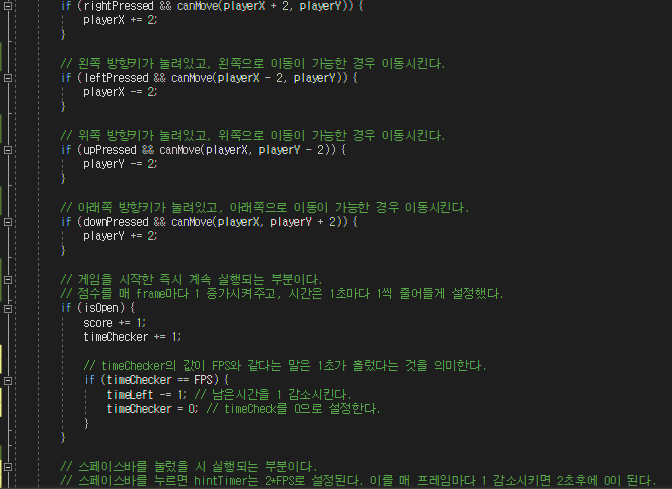
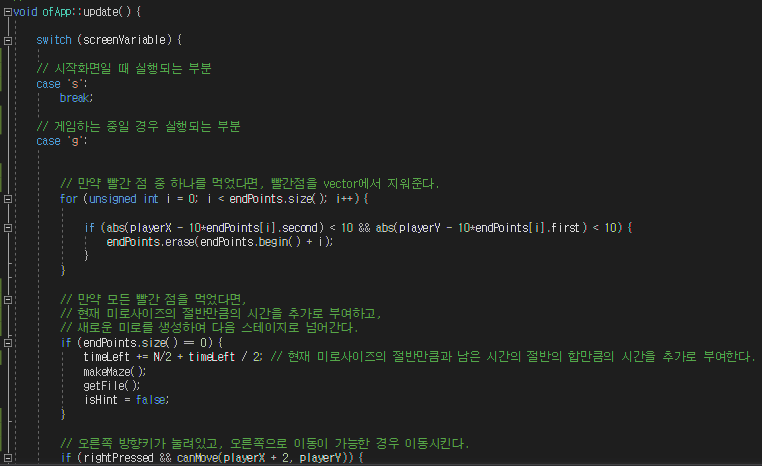
* appMenuFuntion 함수



[사진 7]

[사진 7]은 appMenuFunction 함수를 구성하는 코드를 캡처한 사진이다. 이 함수에서는 메뉴바를 클릭했을 때의 행동을 정할 수 있다. 메뉴바에는 현재 File 이라는 주메뉴와 Exit이라는 부메뉴가 있는데, exit을 클릭하면 프로그램을 종료하도록 설정하였다. 가장 먼저, isexit 변수를 true로 설정하여 프로그램이 종료되었다는 것을 변수로서 저장하고, Vector2File 함수를 호출한다. 이 함수는 프로그램 내부의 ranking\_vector 안에 기록된 점수가 내림차순으로 정리되어 있다. 기록된 점수를 score.txt 라는 외부 파일로 저장해둠으로써 프로그램을 종료하고 다시 실행시켜도 예전의 기록이 남아있도록 했다. 그리고 freeMemory는 프로그램을 작동시키면서 썼던 메모리들을 해제해주는 과정이다. 마지막으로 ofExit함수를 호출하여 오픈프레임워크를 종료한다.

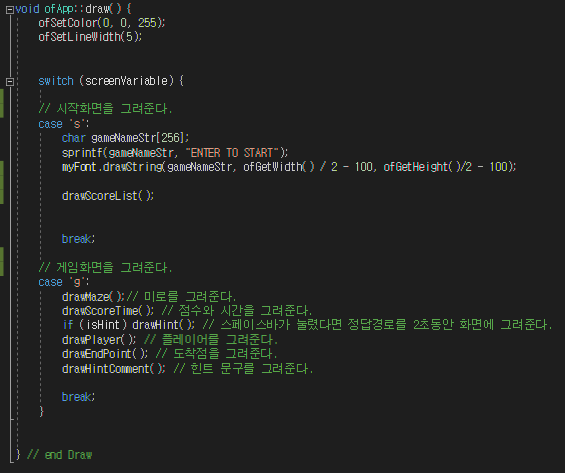
* update 함수



[사진 8]

[사진 8]은 update함수를 구성하는 코드를 캡처한 사진이다. Update 함수는 매 프레임마다 실행되는 함수로써, 지속적으로 변화시켜야 하는 것들을 포함시킨다. 가장 먼저, switch문이 등장하는데, 이는 시작화면과 게임화면에서 해야할 행동이 다르기 때문이다. switch문의 인자로는 screenVariable이 들어가고 “s”는 시작화면, “g”는 게임화면을 의미한다. update함수는 시작화면에서는 따로 할 일이 없다. 별다른 변화가 일어나지 않기 때문이다. 그러나 게임화면에서는 update 함수가 해야할 일이 굉장히 많아진다. 가장 먼저 등장하는 for문은 현재 플레이어의 위치와 endPoints 벡터에 담긴 모든 빨간점 사이의 위치를 비교하여 같은 위치에 있다고 판단 될 경우 빨간점을 삭제하는 반복문이다. endPoints 벡터를 순회하며 현재 플레이어와의 위치관계를 계산해야 하기 때문에 반복문을 이용하였다. 목표는 모든 빨간점을 먹어서 없애야 하는 것이기 때문이다. 그 다음, 만약, 모든 빨간점들을 없앴다면, 즉, 빨간점을 담고있는 vector인 endPoints의 사이즈가 0이라면 현재 미로 사이의 절반과 남은 시간의 절반을 합한 수만큼 추가로 시간을 부여한다. 그리고, makeMaze() 함수를 이용해 새로운 미로를 생성하여 텍스트 파일로 저장하고, getFile()함수를 통해 생성한 텍스트 파일을 불러와서 프로그램 내부의 maze라는 이름의 벡터에 미로정보를 저장한다. 그렇게 한 후, 다음 스테이지로 넘어간다. 물론 isHint 변수는 false로 설정해주어 직전 스테이지에서 힌트를 사용중이었더라도, 그 다음 스테이지로 넘어오면 바로 초기화를 시켜 힌트가 보이지 않도록 한다. 그 다음으로는, 플레이어의 이동이다. 플레이어는 방향키를 이용하여 이동하도록 설정했는데, 방향키를 누른 상태면 pressed 변수가 true가 되도록 설정하였다. 여기서 pressed 변수는 총 4개로, 상, 하, 좌, 우의 방향에 대해 pressed 변수가 각각 존재한다. 그런데 방향키를 눌렀더라도 무조건 이동할 수 있는 것은 아니다. 만약, 이동하려는 부분이 벽으로 막혀있다면 이동할 수 없으므로, 이동가능여부를 사전에 판단해주어야 한다. 이를 판단해주는 함수는 canMove라는 함수로, 뒤에서 다시 설명하도록 하겠다. 그 다음에는 if (isOpen)으로 시작하는 조건문인데, 이는, 미로 파일이 열린 상황에서만 동작하도록 하는 부분이다. 이 부분에서는점수를 매 프레임마다 1점 증가시켜주고, 시간은 초당 1씩 감소하도록 설정하였다. 시간을 초당 1씩 감소하도록 하기 위해서는 timeChecker라는 변수가 추가로 필요하다. 이는, 1초를 세는 변수로서, timeChecker의 값이 FPS와 같다는 것은 1초가 흘렀다는 말과 같다. 따라서, timeChecker의 값이 FPS와 같은 지를 판단하는 조건문을 추가하여, 만약 같다면 1초가 흐른 것이므로 남은 시간을 1 감소시ㅣ고, timeChecker는 다시 0으로 돌려놓는다. 그 이후 if (isHint) 형태로 시작하는 조건문이 등장하는데, 이는 스페이스바를 누르면 실행되는 부분으로서, 플레이어가 힌트를 제공받고 싶을 때 isHint 변수가 true로 바뀌도록 설정하였다. 스페이스바를 누르면 hintTimer는 2\*FPS로 바뀌고 이는 매 프레임마다 1씩 감소하게 된다. 이렇게 진행되다가 hintTimer가 0이 되었다는 것은 2초가 흘렀음을 의미한다. 따라서, 이 때 는 hint finished! 라는 문구를 출력하면서 isHint 변수를 다시 false로 돌려놓는다. 힌트는 2초동안만 제공되도록 계획했기 때문이다. 게임은 시간제한이 있기 때문에 timeLeft가 0이 되면 게임은 종료된다. 이 때는 게임 중에 필요했던 메모리를 해제해주는 freeMemory을 진행하고, 게임이 종료되었으면 시작화면으로 돌아가야 하기 때문에 screenVariable을 “g”로 바꿔준다. 그리고 만약 현재 점수가 최고 점수를 뛰어 넘는 점수라면 최고 점수인 highScore을 현재 점수로 업데이트 해준다. 그리고 registerScore() 함수를 이용해 점수를 저장하는 벡터인 ranking\_vector에 내림차순 정렬이 되도록 가장 처음 항부터 차례로 순회하며 자신보다 더 작은 수가 등장하면 그 자리에 현재 점수를 삽입한다. 별다른 정렬 알고리즘을 사용할 필요 없이 insertion sort에서 사용하는 아이디어와 비슷한 생각을 이용해 매번 정렬된 상태를 유지하도록 했다. 그리고 게임이 끝났으므로 현재 점수를 콘솔 창에 출력해준다.

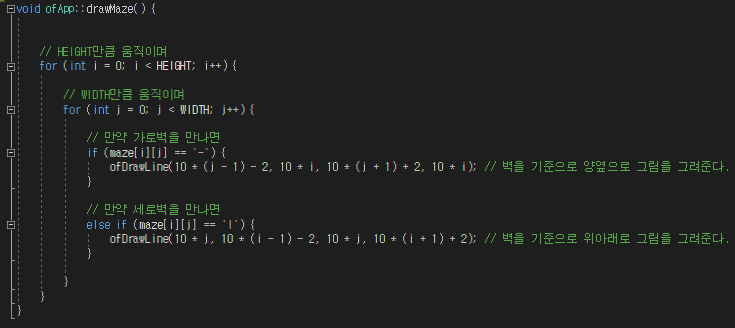
* draw 함수



[사진 9]

[사진 9]는 draw 함수의 코드를 캡처한 사진이다. draw함수 또한 update 함수와 마찬갖;로 매 프레임마다 호출되는 함수이다. 그러나 update함수와는 다르게 draw함수에는 화면에 그리는 과정을 담는 코드를 쓴다. 가장 먼저, 현재 화면이 시작화면인 경우에는 화면 중앙 부분에 “ENTER TO START” 라는 문구를 출력해주어 게임을 시작하려면 엔터키를 누르라는 지침을 준다. 또한, drawScoreList 함수를 호출하는데, 이는 현재 ranking\_vector에 저장되어 있는 기록들 중 top 3의 기록을 출력해주는 부분이다. 자세한 설명은 이 함수에 대해 설명 할 때 하도록 하겠다. 다음으로 현재 화면이 게임화면인 경우인데, 이 때는, 가장 먼저, drawMaze( )함수를 통해 미로를 그려준다. 그 이후, drawScoreTime() 함수를 이용하여 현재 점수와 남은 시간을 화면에 그려준다. 다음으로는 힌트가 눌렸을 때 정답 경로를 2초동안 화면에 그려준다. 다음으로 drawPlayer()을 통해 플레이어의 현재 위치를 그려준다. drawEndPoint는 매 스테이지 마다 랜덤하게 생성되는 빨간점들을 화면 상에 그려주는 역할을 한다. 그리고 drawHintComment() 함수를 통해 미로를 기준으로 살짝 오른쪽 아래부분에 “PRESS SPACE BAR FOR HINT” 라는 문구를 넣어주어, 스페이스바를 누르면 힌트가 화면에 출력된다는 사실을 게임 플레이어에게 알려준다.

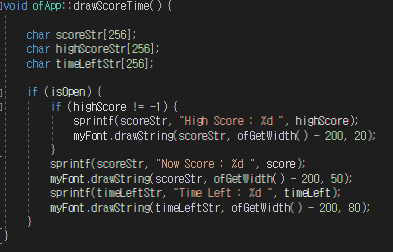
* drawMaze 함수



[사진 10]

[사진 10]은 drawMaze 함수의 코드를 캡처한 사진이다. drawMaze 함수는 미로를 화면에 그려주는 함수이다. 현재 미로는 maze라는 이름의 벡터 자료구조에 저장되어 있다. 이차원 벡터는 아니지만 벡터의 각 원소가 문자열로 저장되어 있어서, 이차원 배열에 접근하듯이 인덱스로 접근할 수 있다. 먼저, 이중 for loop를 이용하여 maze의 원소가 “-“이면 가로벽을 의미하고, 이는 벽을 기준으로 양옆으로 그림을 그려준다. 또한, 만약 maze의 원소가 “|”이면 세로벽을 의마하고, 이는 벽을 기준으로 위아래로 그림을 그려준다. 이런 방식으로 이중 for문이 마무리되면 전체 미로가 그려지게 된다.

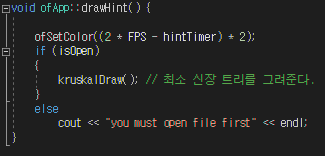
* drawScoreTime 함수



[사진 11]

[사진 11]은 drawScoreTime의 코드를 캡처한 사진이다. 이 함수는 현재 점수와 남은 시간을 화면에 그려주는 함수이다. scoreStr, highScoreStr, timeLeftStr라는 세 개의 문자 배열을 선언한 후, 파일이 열려있는 지의 여부를 확인한다. 만약 파일이 열려있다면, 최고기록이 존재한다면 (최고기록이 -1이 아니라면) “High Score : 최고점수” 형태로 화면에 그려주고 그 이후, “Now Score : 현재점수” 형태로 현재 점수를 화면에 그려주고, 마지막으로 “Time Left : 남은 시간”의 형태로 남은 시간을 화면에 그려준다. 화면에 글씨를 써주기 위해서 sprintf 함수를 이용했다.

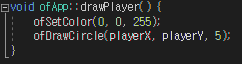
* drawHint 함수



[사진 12]

[사진 12]는 drawHint 함수의 코드를 캡처한 사진이다. 이 함수는 플레이어가 스페이스바를 눌러 힌트를 제공받는다는 의사를 표했을때만 실행되는 함수이다. 먼저, 힌트 경로를 표시해주기 위한 색깔을 지정해야하는데, 검정색에서부터 fade out 되는 효과를 얻고 싶어서 색깔을 매 프레임마다 변경해야겠다는 생각을 했다. 따라서, 색깔의 범위는 검정색부터 현재 배경색까지로 설정하였다. 이는 (2\*FPS - hintTimer)\*2라는 식으로 나타내지고 이를 ofSetColor의 인자로 넣어준다. 이렇게 되면 매 프레임마다 힌트 경로를 그리는 색깔이 바뀌게 될 것이고, 이는 서서히 fade out 되는 효과를 보여줄 것이다. 색깔 선택이 끝났으면 얻어온 힌트경로를 화면에 그려주어야 하는데, 이는 kruskalDraw()함수에서 해준다. 크루스칼 알고리즘을 이용하여 최소신장트리를 얻어 온 다음, 그 경로를 kruskalDraw()를 이용해 그려주는 것이다. 만약, 파일이 열려있지 않다면 “you must open file first”라는 에러 메시지를 출력해주고 함수를 종료했다.

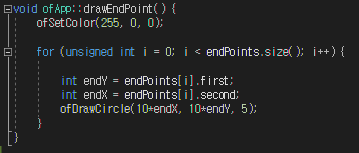
* drawPlayer 함수



[사진 13]

[사진 13]은 drawPlayer 함수의 코드를 캡처한 화면이다. 이 함수는 플레이어의 위치를 파란색 동그라미로 그려주는 함수이다. 따라서, ofSetColor(0,0,255)를 통해 파란색을 얻어냈고 ofDrawCircle(player, player, 5)를 통해 반지름이 5인 원을 x좌표가 playerX, y좌표가 playerY인 지점에 그려준다.

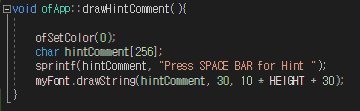
* drawEndPoint 함수



[사진 14]

[사진 14]는 drawEndPoint의 코드를 캡처한 사진이다. drawEndPoint는 각 스테이지마다 빨간점이 총 4개가 등장하고, 그 지점을 모두 방문하여 빨간점을 모두 없앤다면 다음 스테이지로 넘어갈 수 있다. 이를 위해서는 먼저, 빨간 점을 화면에 그려주어야 한다. endPoints라는 벡터는 모든 빨간점의 위치를 저장하고 있다. 이를 순회하며 각각의 점에 대해 반지름이 5인 빨간원을 그려준다.

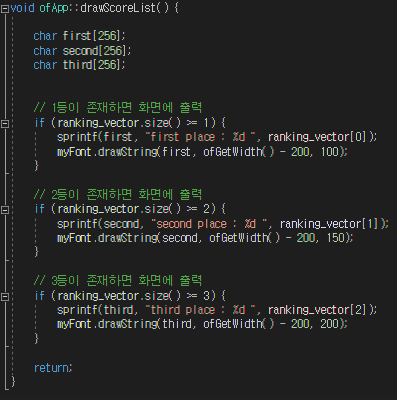
* drawHintComment 함수



[사진 15]

[사진 15]는 drawHintComment 함수의 코드를 캡처한 화면이다. 이 함수는 플레이어에게 스페이스바를 누른다면 힌트를 볼 수 있다는 이야기를 해주기 위해, 미로의 오른쪽 아래 글씨로 “Press SPACE BAR for Hint” 라고 적어주었다. 글씨의 색은 검은색이다. Myfont.drawString함수를 이용해 hintComment에 들어있는 문자열을 화면에 출력해주었다.

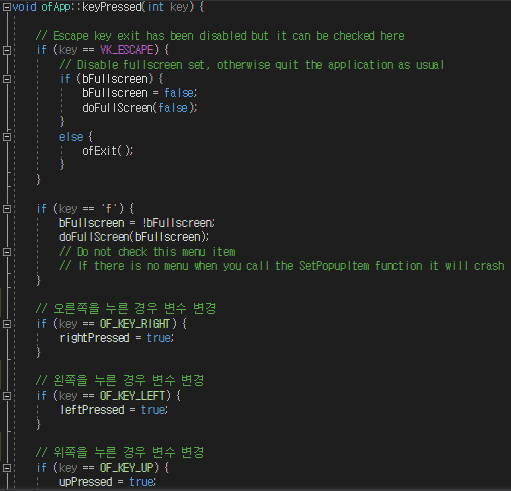
* drawScoreList 함수



[사진 16]

[사진 16]은 drawScoreList 함수의 코드를 캡처한 사진이다. drawScoreList는 게임을 플레이 하다보면 여러 개의 기록이 쌓이게 되고, 그 중에는 top3 또한 분명 존재하게 된다. 따라서, 1등이 존재한다면 “first place : 1등점수” 의 형태로 출력되게 했고, 만약 2등이 존재한다면 그 아래 “second place : 2등점수” 의 형태로 출력되게 했고, 만약 3등이 존재한다면 그 아래 “third place : 3등점수” 의 형태로 출력되게 했다. 모두 myFont의 drawstring 함수를 이용했다.

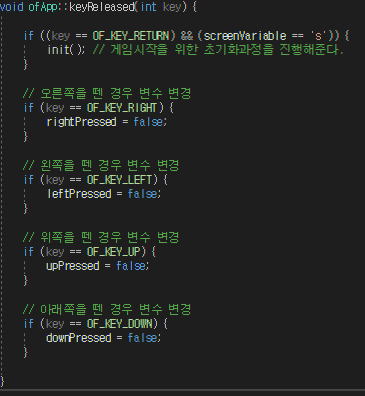
* keyPressed 함수



[사진 17]

[사진 17]은 keyPressed 함수의 코드를 캡처한 사진이다. keyPressed 함수는 키보드의 어떠한 키를 누를 이벤트를 감지하는 함수이다. 가장 먼저, 전체 화면이었을 때 ESC를 누르면 전체화면이 풀리는 것은 이미 구현되어 있다. 또한, “f” 키를 눌렀을 때 전체화면이 되는 것도 구현이 되어 있다. 그 아래는 새롭게 작성한 부분인데, 각 방향키를 눌렀을 때 눌렀다는 것을 감지하고 각 방향의 pressed 변수를 true로 교체하였다. 방향키로 플레이어를 이동시키는 것은 pressed 변수를 두지 않고 클릭한다면 바로 플레이어의 위치를 직접 변경시켜도 된다. 그러나, 윈도우 키보드는 어떠한 키를 꾹 누르더라도, 한번만 눌린 키가 출력되고 잠시 있다가 연속해서 쭉 출력되는 특성을 가지고 있다. 이러한 키보드의 특성 때문에 프로젝트 특성상 방향키의 전환이 매우 빈번한 상황에서 이러한 상황이 발생한다면 원활하게 게임을 플레이하지 못하고 자꾸 끊기는 것을 경험할 것이다. 이는 플레이어의 피로도를 급격하게 높이며, 게임의 흥미 또한 잃게 만든다. 그리고, 각 if문에서 직접적으로 플레이어의 위치를 바꾸면 안 되는 또 다른 이유는 여러키를 한번에 인식하지 못하는 점이 있다. 동시에 여러 키를 눌러서 이동하는 상황도 게임에서 필요하다. 대각선 방향으로 가는 것이 시간단축에 효율적인 경우가 미로를 탈출하는 데 등장하기 때문이다. 이러한 경우를 커버하기 위해서 pressed 변수를 따로 두는 것이다. 이를 두면 여러 키를 누르게 되더라도 동시에 두 개 이상의 pressed 변수가 변화하는 것이고, pressed가 변화했다면 플레이어의 위치는 동시에 여러방향으로 바뀔 수 있기 때문이다. 바로 아래에서 설명할 keyRealesed 함수에서는 키보드를 떼는 이벤트를 감지하는데, pressed 변수는 누르고 있을 때는 true였다가 떼자마자 false로 바뀌도록 할 것이다. 이렇게 프로그램을 작성하면 끊김없이 원활하게 게임을 플레이 할 수 있다. 그 아래 스페이스바를 눌렀을 때 해야할 행동을 구현한 부분이 있다. 먼저, isHint를 true로 설정한다. 그리고 hitnTimer를 2\*FPS로 설정해주어 힌트는 2초동안만 화면에 출력되도록 한다. 그리고 edgeCount와 heapCount라는 변수를 새롭게 두었다. 먼저, edgeCount는 간선이 노드개수 - 1개가 될때 까지 크루스칼 알고리즘을 돌리기 위해 만든 변수이다. 또한, heapCount는 현재 힙에 들어있는 원소의 개수를 저장하는 변수이다. 그 이후 union-find 알고리즘을 수행하기 위해 부모 노드를 저장하는 배열인 parent배열을 동적할당으로 정의하였다. 또한 모두 -1로 설정해주어 초기화를 해준다. 이후, 힙 자료구조와 parent 배열이 준비가 되었으므로, kruskal() 함수를 호출하여 크루스칼 알고리즘을 이용하여 모든 빨간점들까지의 최단경로를 구하여 PATHS 벡터에 저장한다.

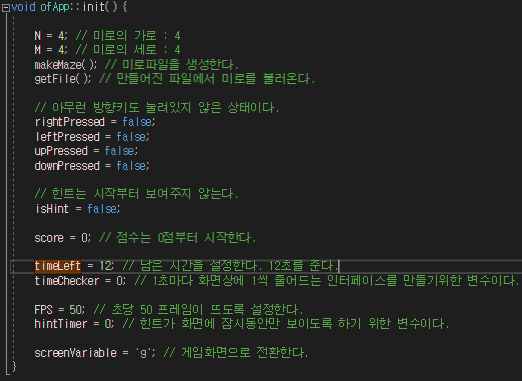
* keyReleased 함수



[사진 18]

[사진 18]은 keyReleased 함수의 코드를 캡처한 사진이다. keyReleased 함수는 keyPressed와 유사하지만 다른점은 keyPressed는 키를 누르는 이벤트를 감지하는 함수라면 keyReleased는 누른 키를 떼는 이벤트를 감지하는 함수이다. 먼저, 엔터키를 눌렀다 떼게 되면 게임이 시작되는데, 게임이 이미 시작된 상황에서 다시 엔터 키를 누르는 경우에는 게임이 새로 시작되면 안된다. 따라서, 이를 위해 엔터키를 누르고 screenVariable이 ‘s’인 경우에만 엔터키를 누른 동작이 인식되도록 하였다. 엔터 키를 누른다면 init()함수가 호출되는데, 이는 게임을 시작하기 위한 여러 변수들의 초기화과정을 담고있다. 그 이후 4개의 조건문이 등장하는데 모두, 방향키를 떼면 true였던 pressed변수들을 다시 false로 변환해주는 과정이다.

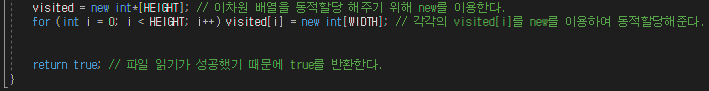
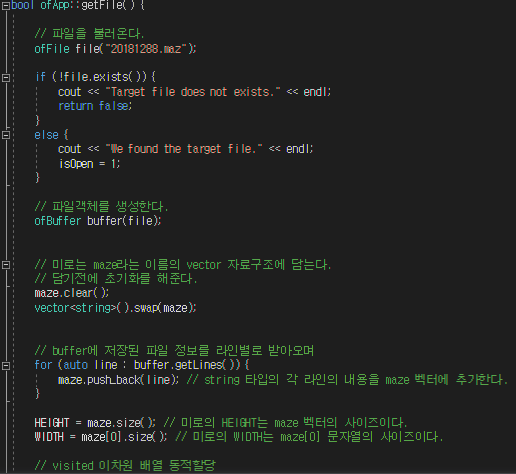
* init 함수



[사진 19]

[사진 19]는 init함수의 코드를 캡처한 사진이다. 이 함수는 시작화면에서 엔터를 눌러서 게임화면으로 넘어올 때 실행되는 함수이다. 가장 먼저, 미로의 가로 세로를 모두 4로 설정해주고, makeMaze()함수를 이용해 미로를 생성한 후 텍스트 파일에 저장한다. 그 이후, getFile()을 이용하여 미로가 저장되어 있는 텍스트 파일을 불러와 maze라는 이름의 벡터에 미로를 저장한다. 그리고 시작하자마자는 아무런 방향키도 눌러있지 않은 상태이므로 rightPressed, leftPressed, upPressed, downPressed 모두 false로 초기화한다. 그리고 힌트도 시작부터 보여주지 않으므로 isHint 또한 false로 설정해준다. 그리고 처음 점수는 0점부터 시작하기 때문에 score은 0으로 초기화하고, 남은 시간을 저장하는 timeLeft는 12로 설정한다. 1초마다 화면상에 1씩 줄어드는 인터페이스를 만들기 위한 변수인 timeChecker는 0으로 초기화한다. 그리고 FPS는 50으로 설정해준다. 초당 50프레임이 뜨도록 설정한다. 그리고 hintTimer는 힌트가 화면에 2초동안만 보이도록 하기 위한 변수이다. 그리고 게임화면으로 들어가야 하기 때문에 screenVariable을 ‘g’로 설정해준다.

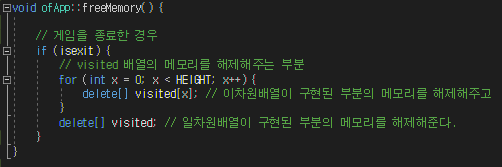
* getFile 함수



[사진 20]

[사진 20]은 getFile함수의 코드를 캡처한 화면이다. getFile함수는 makeMaze 함수에 의해 만들어진 .maz 파일을 불러와서 maze라는 벡터 자료구조에 미로정보를 저장하는 함수이다. 먼저, ofFile 객체를 이용하여 20181288.maz 파일을 불러온다. 그리고 buffer를 생성한다. 미로는 maze라는 이름의 벡터 자료구조에 담아야 하므로, 담기전에 초기화를 해준다. 그 이후 buffer에 저장된 파일 정보를 라인별로 받아오며 string 타입의 각 라인의 내요을 maze 벡터에 추가한다. 그리고 HEIGHT와 WIDTH는 가로 세로를 의마하고 각각 maze의 벡터 사이즈이고, maze[0]의 벡터 사이즈임을 알고 있으므로 그렇게 구한다. 그 다음 크루스칼 알고리즘을 진행하기 전에 각 간선의 cost를 계산하기 위한 dfs함수를 돌리는 과정이 있는데, 이 때 이용하는 dfs 알고리즘에는 visited 배열이 쓰인다. 방문처리를 해주면서 그래프를 탐색하는 것이 dfs 알고리즘의 핵심내용이다. 그 이후 모든 과정이 성공적으로 마무리 되었으면 파일 읽기가 성공했기 때문에 true를 반환한다.

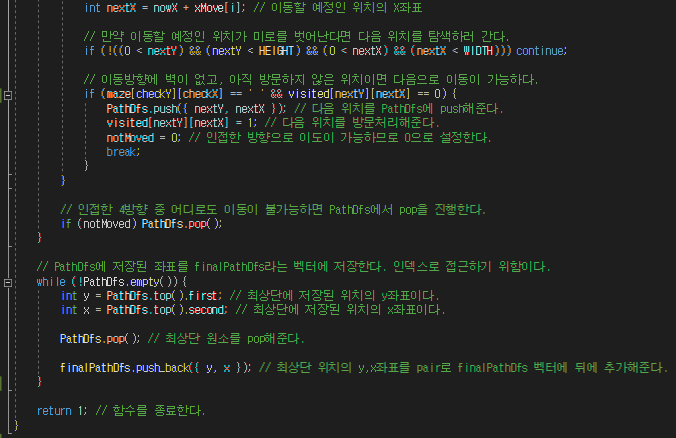
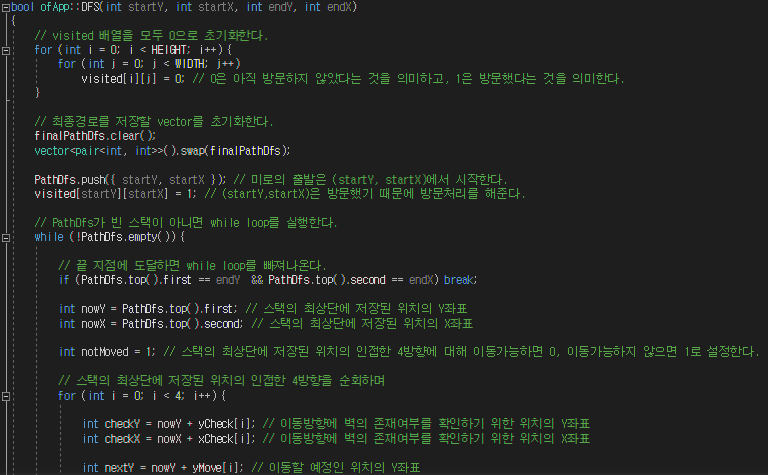
* freeMemory 함수



[사진 21]

[사진 21]은 freeMemory함수의 코드를 캡처한 사진이다. freeMemory함수는 프로그램이 종료하면 사용했던 메모리들을 모두 해제해주어야 하는데, 그과저을 한번에 진행하는 함수이다. 현재 프로그램에서는 동적할당된 배열은 visited 배열이 있다. 이 배열은 바로 위의 getFile 함수에서 동적할당되었다. 프로그램이 종료되면 이 배열의 메모리는 해제되어야 한다. 이차원 배열이기 때문에 이차원 메모리를 해제하는 방식으로 이차원 배열이 구현돈 부분의 메모리를 해제해주고, 일차원 배열이 구현된 부분의 메모리를 해제해준다.

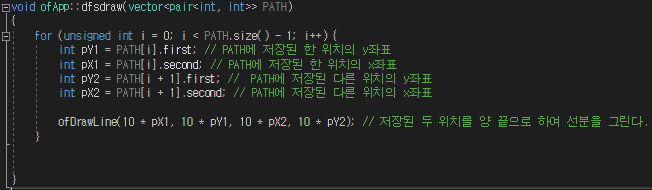
* Dfs 함수



[사진 22]

[사진 22]는 dfs함수의 코드를 캡처한 사진이다. Dfs 함수는 maze project를 진행할 때 구현해놓은 코드의 일부분을 이용하였고, 변경한 부분만 설명하도록 하겠다. 먼저, 실습에서는 dfs를 가장 왼쪽 위를 시작으로 하고, 가장 오른쪽 아래를 끝으로 하여 알고리즘을 돌렸었다. 따라서, 우리가 얻게 되는 경로는 시작점에서 끝 지점까지의 경로만 알 수 있었다. 그러나, 이를 변형하여, 임의의 두 지점 사이의 경로를 얻을 수 있는 함수로 변화시켰다. 따라서, 함수의 인자는 총 4개가 들어가는데 시작점의 x좌표 시작점의 y좌표, 끝 점의 x좌표, 끝 점의 y좌표가 그것이다. 그렇게 한 후, 기존의 함수에서의 시작점 (1,1)을 대입했던 자리에 startX, startY를 대입하고 기존의 함수에서 끝점을 대입했던 자리에 endX, endY를 대입한다. 또한, 실습에서는 탈출 경로뿐만 아니라 dfs 알고리즘이 돌아가는 과정에서 방문했던 모든간선들을 출력하라고 했는데, 이는 기말 프로젝트에서는 필요없는 과정이고 메모리만 차지한다고 생각이 들어서 그 부분에 대한 배열은 삭제하였다. 이런식으로 함수를 작동시키면 결과적으로 finalPathDfs에는 현재 인자로 넣어 준 두 점사이를 연결하는 경로가 저장될 것이다.

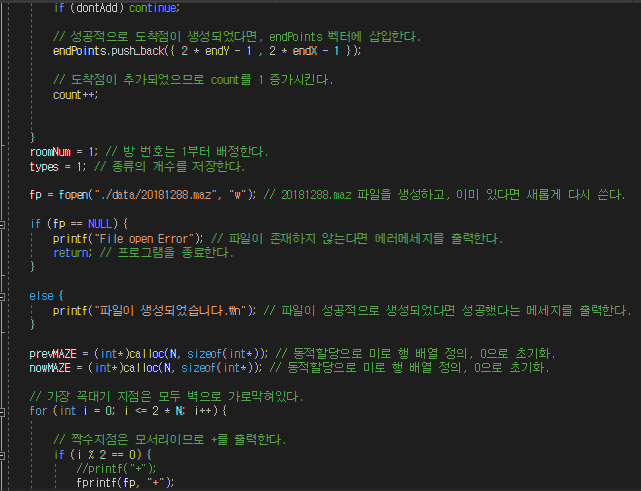
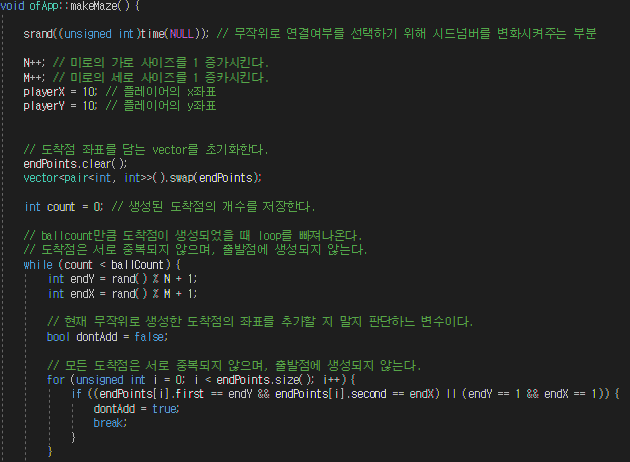
* dfsdraw 함수



[사진 23]

[사진 23]은 dfsdraw함수의 코드를 캡처한 사진이다. dfsdraw에서는 실습에서와 다르게 경로를 하나 받게 설정하였다. Vector 자료구조안의 pair이 들어있는 경로를 인자로 받은 후 그 인자들을 하나씩 순회하며 각각의 경로를 선분으로 연결해서 그린다. 그냥 점들을 연결하면 잘 보이지 않아서 각각의 좌표에 10을 곱해서 그렸다.

* makeMaze 함수



[사진 24]

[사진 24]는 makeMaze 함수의 일부를 캡처한 사진이다. makeMaze함수는 먼저 빨간점의 위치를 정해준다. 빨간점은 총 4개가 생성되도록 했고, 모두 다른위치에 생성되도록 했다. 또한, 빨간점은 시작점의 위치아는 겹치지 않도록 했다. 만약, 겹친다면 시작하자마자 빨간점 하나를 제거하고 게임을 시작하는 상황이 발생할 수 있기 때문이다. 빨간점을 선택하기 위한 제한조건을 만족시키기 위해 위의 사진에서 여러 개의 for문과 if문이 등장하는 것이다.

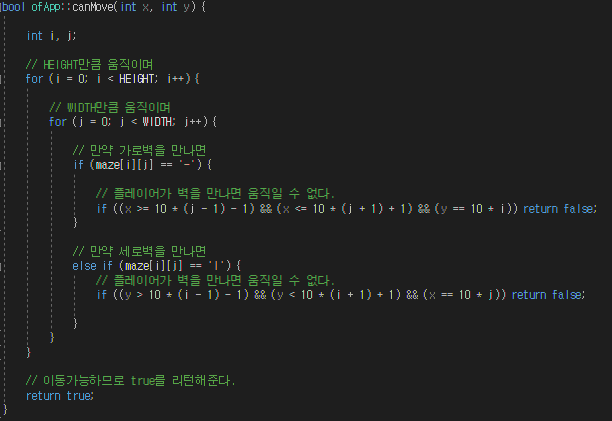
* horizontalCheck 함수

이 함수는 실습에서 엘러 알고리즘을 구현할 때 수평방향의 벽을 설치하는 과정을 담고있는 함수를 그대로 사용한 것이다.

* verticalCheck 함수

이 함수는 실습에서 엘러 알고리즘을 구현할 때 수직방향의 벽을 설치하는 과정을 담고있는 함수를 그대로 사용한 것이다.

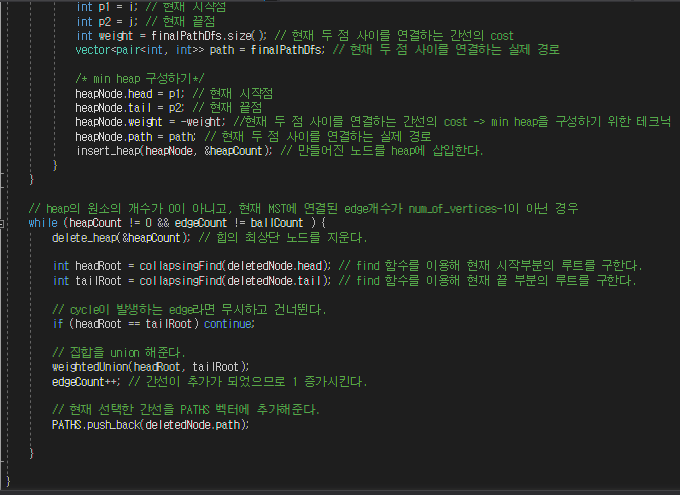
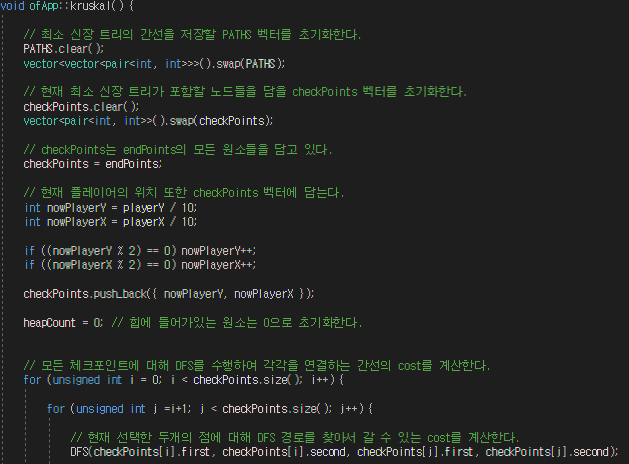
* canMove 함수



[사진 25]

[사진 25]는 canMove함수의 코드를 캡처한 사진이다. canMove함수는 플레이어가 이동을 할 때 다음위치로 이동이 가능한지 여부를 판단해주는 함수이다. 만약, 이동이 불가능한 위치라고 판단하면 방향키를 누르고 있더라도 이동하지 않게 된다. 먼저, 이중 for문을 돌며 플레이어가 가로벽을 만나면 움직일 수 없도록 작성하였고, 세로벽을만나더라도 움직일 수 없도록 코드를 작성했다. 벽의 위치와 플레이어의 위치가 겹치면 움직일 수 없도록 했다.

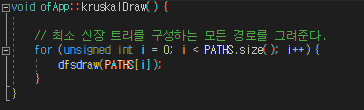
* kruskal 함수



[사진 26]

[사진 26]은 kruskal 함수의 코드를 캡처한 사진이다. 이번 프로젝트에서 가장 핵심이 되는 부분이라고 생각한다. 이번 프로젝트에서는 시작점에서 파란공이 움직이며 빨간점들을 모두 먹으면 다음 스테이지로 넘어가게 되는데, 매 스테이지마다 미로의 사이즈는 점점 커지게 된다. 여러 스테이지를 거치게 되면 미로찾기가 굉장히 어려워지는 순간이 오는데, 그러한 플레이어의 의욕을 상실하지 않게 하기 위해 힌트를 제공한다. 힌트는 현재 플레이어의 위치에서 남은 빨간점들을 방문할 수 있는 최단 경로를 추천해준다. 이 최단경로는 여러 개의 점을 지나는 최소 신장 트리이고 이것을 구하는 과정이 이 함수안에 담겨있다. 먼저, 최소 신장 트리를 저장할 PATHS 배열을 초기화한다. 또한, checkPoints라는 새로운 벡터를 초기화하는데, 여기에는 현재 플레이어의 위치와 남은 빨간점들의 위치가 모두 들어간다. 그렇게 하고 크루스칼 알고리즘을 진행하기 위해서는 각각의 점들 사이의 거리를 모두 알아야 하기 때문에, 두개의 점을 선택하고 dfs함수를 돌려서 둘 사이의 경로의 길이를 계산한다. 이렇게 계산을 한 이후에는 하나의 노드에 정보들을 집어넣는데, 먼저 시작점과 끝점, 그리고 두 점 사이의 경로의 길이 그리고 마지막으로 실제 경로를 집어넣는다. 그리고, 힙 자료구조에 넣는데 min heap을 구현하였다. 그리고 min heap의 기준이 되는 변수는 두 점 사이의 경로의 길이로 설정했다. 이러한 과정을 거쳐 모든 간선들을 힙에 집어넣었다면, 힙 자료구조의 특성상 제거 매서드는 시간복잡도 O(1)에 가능하다. 따라서, 힙에서 최상단 노드를 뽑은 후, 그렇게 뽑은 간선을 실제 최소 신장 트리를 구성하는 간선으로 집어넣는다. 그러나, 최소신장트리는 사이클이 생기면 안된다. 사이클 판단을 하기 위해서는 find와 union 알고리즘이 필요하다. 그러나, 효율성을 위해서 경로압축기법을 이용한 find와 균형잡힌 트리의 구성을 도와주는 weightedUnion함수를 활용하였다. 이렇게 해서 최소 신장 트리를 얻게 되면, 그 경로는 현재 플레이어의 위치에서 모든 빨간점까지 갈 수 있는 최단 경로를 얻게 된다. 이렇게 얻은 경로는 PATHS 벡터에 저장된다.

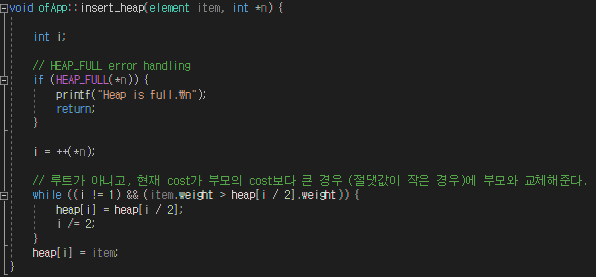
* Kruskaldraw 함수



[사진 27]

[사진 27]은 kruskalDraw함수의 코드를 캡처한 사진이다. kruskalDraw는 kruskal 함수에 의해 얻은 최소신장트리를 화면에 출력해주는 과정이다. 최소 신장트리의 각 원소는 총 4개의 점을 가지고 있는데, 시작점과 끝점의 x, y 좌표들이다. 따라서, 이 점들을 연결한 직선을 그려주는 함수인 dfsDraw함수를 호출하여 최소신장트리를 그린다.

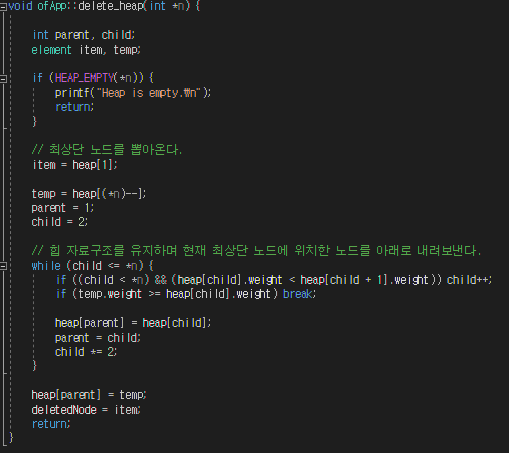
* Insert\_heap 함수



[사진 28]

[사진 28]은 insert\_heap 함수의 코드를 캡처한 사진이다. Insert\_heap은 min heap의 구조를 변화시키지 않고 원소를 삽입시키는 함수이다. 먼저, 힙이 꽉 차있다면 더 이상 원소를 추가할 수 없으므로, HEAP\_FULL 매크로를 이용해 이를 확인하고 원소삽입이 불가능하다면 에러메세지를 출력해준다. 그리고, 원소 삽입은 루트가 아니고, 현재 cost가 부모의 cost보다 큰 경우(절댓값이 작은 경우)에 부모와 교체해준다. 매순간 이 과정을 거쳐서 힙을 구성한다면 언제나 min heap이 완성되어있다.

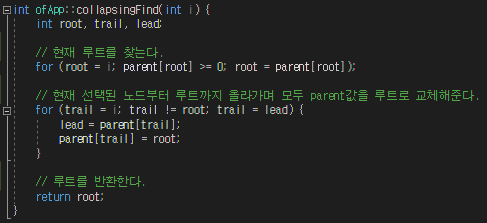
* delete\_heap 함수



[사진 29]

[사진 29]는 delete\_heap 함수의 코드를 캡처한 사진이다. Delete\_heap은 힙 자료구조에서 원소를삭제하는 과정을 담은 함수이다. 힙 자료구조 특성상 원소를 삭제하는 것은 최상단 노드를 삭제하는 것이다. 따라서, complete binary tree의 가장 마지막 원소를 제일 위로 올린 후, 힙 자료구조를 유지하며 현재 최상단 노드에 위치한 노드를 아래로 내려보낸다. 더 이상 내려올 곳이 없다면 그곳에서 멈추면 된다. 매번 이러한 과정ㅇ르 거치면 힙 자료구조에서의 원소 삭제는 어렵지 않다. 그런데 만약 힙이 비어있는 상황에서도 원소를 삭제하려고 시도한다면 오류가 발생할 것이다. 따라서, 이를 방지하기 위해 HEAP\_EMPTY 매크로를 이용하여 힙이 비어있는 지의 여부를 판단한다. 만약 비어있는데 원소를 삭제하려고 한다면 에러메세지를 출력하고 함수를 종료한다.

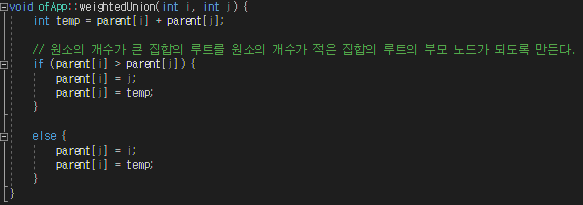
* CollapsingFind 함수



[사진 30]

[사진 30]은 collapsingFind함수의 코드를 캡처한 사진이다. collapingFind는 일반적인 find에서 조금 더 개선한 방법이다. 크루스칼 알고리즘의 효율성을 극대화하기 위해 사용된 함수이다. 일반적인 find 함수는 호출을 하게 되면 현재 호출한 원소의 부모를 쭉쭉 타고 올라가면서 가장 위쪽에 위치한 부모노드를 반환해주게 된다. 그러나, 만약, 같은 원소를 다시 불렀다면 다시한번 부모를 쭉쭉 타고 올라가야 하는 과정이 반복된다. 따라서, 이러한 올라가는 과정을 한번만 하자는 아이디어로 이 함수가 작성되었다. 가장 먼저, 현재 루트를 찾는다. 그리고 현재 선택된 노드부터 루트까지 올라가며 모두 parent값을 루트로 교체해준다. 그리고 루트를 반환한다. 따라서, 현재 궁금했던 노드가 아닌 다른 노드의 부모 또한 한번에 교체를 해주어 그 다음 탐색때 굉장히 빠른 속도로 find함수가 작동할 수 있게 된다. 이를 경로압축기법이라고 한다.

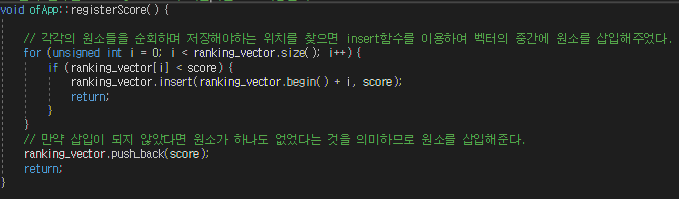
* WeightedUnion 함수



[사진 31]

[사진 31]은 WeightedUnion함수의 코드를 캡처한 사진이다. 이 함수 또한 collapsingFind와 마찬가지로 크루스칼 알고리즘의 효율성을 극대화하기 위한 함수이다. 우리는 크루스칼 알고리즘에서 사이클 판별을 하기 위해 같은 집합인지 확인하게 된다. 이는 find함수를 이용하는 것이고, 만약 다른 집합임이 확인되었으면, 이 두 집합을 하나로 합쳐야 한다. 두 집합을 하나로 합치는 것은 하나의 집합의 루트가 다른 집합의 루트의 부모가 되는 것이다. 그러나, 아무 제한조건없이 집합을 합치게 되면 balanced 되지 않은 트리가 만들어 질 수도 있다. 트리의 효율성은 트리의 balanceness에 비례하게 된다. 따라서, 그냥 union을 하게 되면 매우 비효율적으로 코드가 작동 될 위험이 있다. 따라서, WeightedUnion 함수를 이용한다. 이 함수는 두 집합 중 더 원소의 개수가 적은 집합이 큰 집합의 자식이 되도록 만든다. 이 방법을 이용하면 그냥 하는 것보다 훨씬 균형잡힌 트리가 만들어진다.

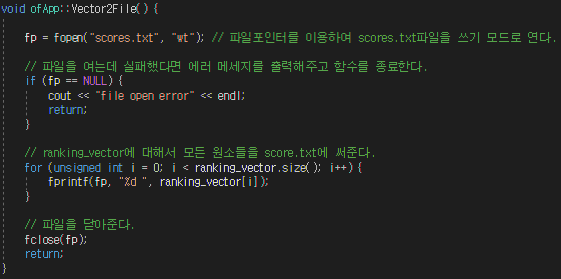
* registerScore 함수



[사진 32]

[사진 32]는 registerScore 함수의 코드를 캡처한 사진이다. 이 함수는 현재 기록된 점수를 모든 점수가 저장되어 있는 벡터인 ranking\_vector의 적절한 위치에 삽입해주는 함수이다. 이 때, 적절한의 의미는 삽입 이후에 ranking\_vector은 항상 내림차순정렬 상태를 유지해야 한다. 이를 위해서는 벡터 내부의 원소들을 순회하며 자신보다 더 작은 원소를 발견했다면 그 자리에 삽입해주면 된다. 만약, 삽입이 되지 않았다면, 벡터의 크기가 0이라는 의미이므로, push\_back함수를 이용해 원소를 삽입해준다.

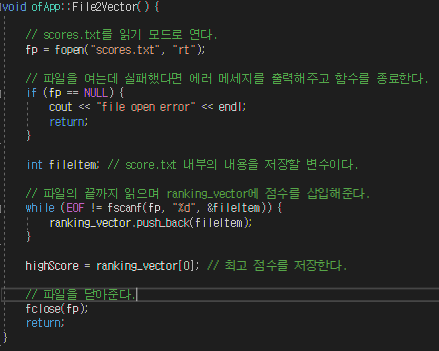
* Vector2File 함수



[사진 33]

[사진 33]은 Vector2File함수의 코드를 캡처한 사진이다. 이 함수는 프로그램상의 점수를 저장하는 ranking\_vector의 내용을 프로그램이 종료된 이후에도 보존하고 싶어서 파일 형태로 보존해야겠다는 생각을 함수로 옮겼다. 먼저, 파일포인터로 scores.txt 파일을 연다. “wt” 모드로 열었기 때문에 파일이 존재한다면 내부의 내용이 다 삭제되고 파일이 존재하지 않는다면 파일이 생성된다. 이렇게 한 이후에 scores.txt 파일에 fprintf 함수를 이용하여 점수를 파일에 출력하고, 모든 출력이 완료되었으면 파일을 닫아준다.

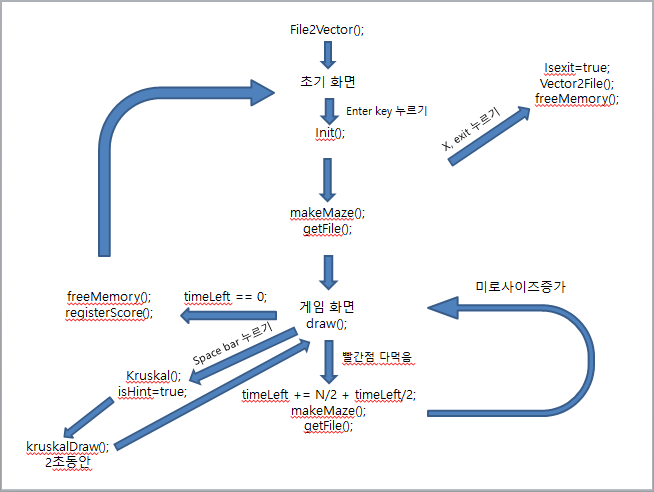
* File2Vector 함수



[사진 34]

[사진 34]는 File2Vector 함수의 코드를 캡처한 사진이다. 이 함수는 score.txt 내부의 점수정보들을 프로그램 상의 ranking\_vector로 이동시키는 과정을 하는 함수이다. 만약, score.txt가 존재하지 않는다면 에러메세지를 출력한다. 만약 score.txt가 존재한다면 파일의 끝까지 읽으면서 각 정수를 모두 ranking\_vector에 push\_back 해준다. 그리고 최고 기록(highScore)은 ranking\_vector은 내림차순 정렬 되어있으므로 가장 앞 원소가 최고 점수에 해당하고 모든 과정을 끝냈다면 파일을 닫아준다.

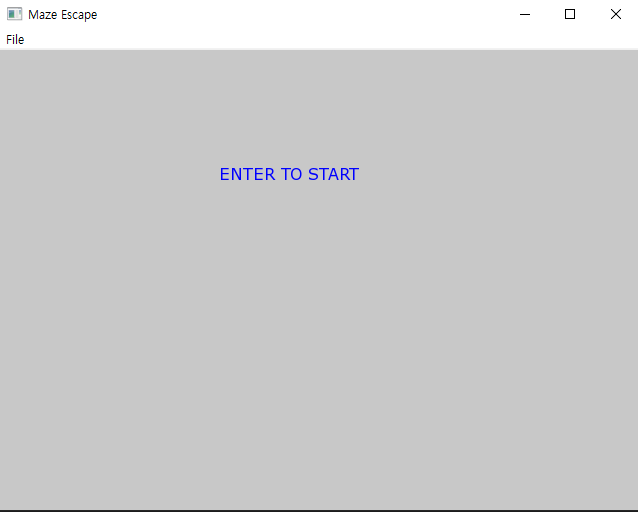
1. <프로젝트 플로우차트 첨부 및 자료구조 알고리즘, 시간/공간 복잡도>



[사진 35]

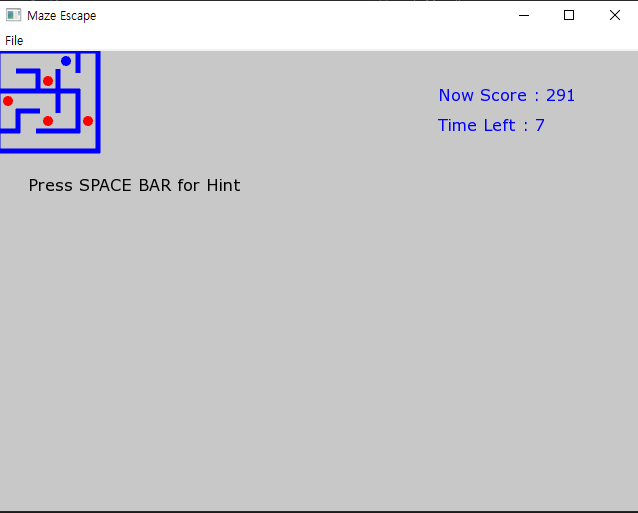
[사진 35]는 이번 기말 프로젝트의 전체적인 플로우차트이다. 가장 먼저, File2Vector(); 함수를 호출하는데, 이것은 score.txt의 내용을 ranking\_vector이라는 벡터에 옮기는 과정이다. 이 과정은 벡터의 크기가 N이라고 하면 시작복잡도와 공간복잡도는 모두 O(N)이다. 그 이후, 초기화면으로 왔을 때는 아무런 시간복잡도나 공간복잡도가 사용되지는 않는다. 이제, 엔터 키를 누름과 동시에 프로젝트의 전체적인 과정이 드러난다. 엔터키를 누르게 되면 init() 함수가 호출된다. init함수에는 미로 탈출 게임을 진행할 초기화과정이 진행되는데, 미로의 사이즈, 현재 점수, 방향키가 눌러진 것의 여부 등등을 초기화한다. 이 과정은 반복문을 도는 과정이 없고 오로지 값 대입만으로 이루어진 함수이므로 시간복잡도 및 공간복잡도는 O(1)이다. 그 이후, makeMaze()함수와 getFile 함수가 호출된다. 먼저, makeMaze 함수는 빨간점 선택을 가장 먼저한다. 빨간점은 현재 4개 존재하도록 프로그래밍 했기 때문에 while loop를 돌며 빨간점을 4개 선택했다면 while loop을 종료한다. 이 while loop은 꼭 4번만 도는 것은 아니다. 왜냐하면 중복된 지점의 빨간점이 선택될 수도 있고, 시작점이 빨간점으로 선택될 수도 있기 때문이다. 이러한 과정을 모두 거치면 while loop를 탈출한다. 이 과정에서 while loop는 공의개수인 ballCount에 비례한다. 즉, 시간복잡도는 O(ballCount)이다. 또한 ballCount만큼의 빨간점의 위치를 저장해야 하므로 공간복잡도 또한 O(ballCount)이다. 그 이후, 미로를 생성하는데, 미로는 엘러 알고리즘을 이용해 생성한다. 엘러 알고리즘은 현재 prevMaze와 nowMaze만을 사용하여 생성하고 있다. 이 두개의 배열을 가지고 가장 꼭대기 지점을 출력하는 부분에 의해 시간복잡도가 결정이 된다. 미로의 가로를 N이라고 하면 엘러 알고리즘의 시간복잡도는 O(2\*N) = O(N)이고, 공간복잡도 또한 O(2\*N) = O(N)이다. 따라서, makeMaze 함수의 시간복잡도 및 공간복잡도는 O(ballCount + N) 이다. 그 이후 getFile함수를 호출한다. getFile 함수는 makeMaze 함수에 의해 .maz 파일이 만들어 졌고, 그렇게 만들어진 파일을 받아와서 미로정보를 이차원 배열에 저장하는 것이다. 파일객체를 이용해 .maz 파일에 접근한 후, for문을 돌면서 모든 행의 정보들을 하나의 문자열로 가져와서 저장한다. 이 과정은 만약 미로의 가로를 N, 미로의 세로를 M이라고 한다면 O(M)만큼의 시간복잡도가 걸린다. 모든 행에 대해서 반복문을 돌리고 있기 때문이다. 그치만 미로의 정보를 저장하는 maze 벡터는 공간복잡도를 O(N\*M)이라고 이야기할 수 있다. 원소로 문자열을 저장하지만 결국 원소의 개수는 각 행마다 N개이기 때문이다. 그 이후, visited 배열을 동적할당 해주는데, 이는 시간복잡도가 O(M)만큼 걸린다. 모든 행에 대해서 각각 메모리를 할당해주고 있기 때문이다. 그러나 공간복잡도는 이차원 배열이기 때문에 O(N\*M)이라고 이야기 할 수 있다. 따라서, getFile 함수의 전체적인 시간복잡도는 O(M+M) = O(M)이고, 공간복잡도는 O(N\*M + N\*M) = O(N\*M)이다. 이 과정을 거치고 나면 게임화면으로 들어오게 된다. 이 상황에서 스페이스바를 누르게 되면 크루스칼 알고리즘에 의해 최소 신장 트리가 찾아진다. 최소 신장 트리를 생성하기 위해서는 일단 현재 남아있는 모든 빨간점과 플레이어의 위치만큼의 개수에 대해 생각해보아야 한다. 이는 ballCount에 비례하는 수일 것이다. 따라서, 이 점들 중 두개의 점을 선택하는 것의 시간복잡도는 O(ballCount^2)이다. 그리고 dfs 함수를 돌릴 때의 시간복잡도를 생각해보면, 먼저, visited 배열을 초기화하기 위해 도는 이중 반복에 의해 O(N\*M)이다. 그 이후 dfs 알고리즘이 진행되는데, 각 노드마다 4번의 탐색이 진행되므로 인접리스트 방식의 그래프 탐색이라고 봐야 한다. 최악의 경우 모든 노드마다 4번의 탐색이 진행될 수도 있음을 의미하고 이는 시간복잡도 측면에서 O(4\*M\*N) = O(N\*M)이다. 따라서, dfs 함수를 돌리는 과정에서의 시간복잡도는 O(N\*M+ N\*M) = O(N\*M)이다. Dfs 함수의 공간복잡도는 현재 함수에서 쓰이는 자료구조는 PathDfs라는 스택과 finalPathDfs라는 벡터이다. 이 둘은 모두 O(N\*M)만큼의 공간복잡도를 가진다. 모든 지점을 경로로 가지는 경우가 존재할 수도 있기 때문이다. 그 이후, 각각의 dfs과정에 의해 힙 자료구조에 하나의 원소가 추가가 된다. 현재 간선의 개수는 edgeCount로 설정하고 있으므로, 힙 자료구조에 추가하는 과정에서의 시간복잡도는 O(log(edgeCount))이다. 공간복잡도는 edgeCount만큼의 노드가 들어가 있으므로 O(edgeCount)이다. 만약, 힙 자료구조를 사용하지 않았다면 지금과 같이 로그스케일의 시간복잡도가 나올 수 없고, 매번 최소인 간선을 선택하기 위해 자료구조 전체를 훑어야 하므로, 제곱의 시간복잡도가 나왔을 것이다. 그부분에서 힙 자료구조를 이용하는 구현은 시간복잡도 측면에서 굉장히 효율적이다. 그 이후, 힙 자료구조에 들어가있는 간선들을 뽑아내면서 사이클이 생기지 않는 상황에서 PATHS에 추가해서 최소 신장트리를 구한다. 일단 힙 자료구조에서 삭제하는 과정 또한 O(log(edgeCount))이다. 그리고, find함수의 시간복잡도 및 공간복잡도를 생각해봐야 하는데, 현재 경로 압축기법이 사용된 find 함수를 사용하고 있으므로, find 함수의 평균적인 시간복잡도는 O(1)이라고 볼 수 있다. 지나간 노드의 부모는 모두 루트를 가리키고 있으므로 평균적으로 find함수를 부르자마자 루트노드를 반환해줄 확률이 높다.이 부분에서 그냥 find 함수를 쓰는것보다 훨신 효율적임을 알 수 있다. find함수의 공간복잡도는 이미 존재하는 트리를 활용하는 것이므로 O(edgeCount)라고 쓸 수 있다. 또한, union 과정에서는 균형잡힌 트리를 활용하기 위해 WeightedUnion 함수를 사용한다. 이를 사용하면 find 과정을 진행하는데 더더욱 시간이 단축된다. Leaf node에서 타고 올라가야할 노드가 균형잡힌 트리이기 때문에 O(log(edgeCount))의 시간복잡도를 가진다. 따라서, 실제 루트노드를 찾는 시간은 O(1)과 O(log(edgeCount))의 사이 시간정도가 걸리겠지만 편의상 O(1)로 쓴다. 한번에 찾아지는 경우가 굉장히 많아 질 것이기 때문이다. 이러한 함수들의 특성으로 보았을 때, 시간복잡도 및 공간복잡도를 효율적으로 줄 일 수 있는 과정임을 알 수 있다.

1. 기말 프로젝트에서는 미로 탈출 게임을 제작하였다. 게임은 엔터 키를 누르면 시작하게 되고, 시작은 4X4 크기의 무작위 미로가 생성되어 화면에 출력된다. 또한, 도착점의 역할을 하는 빨간점들이 4개 있다. 목표는 매 스테이지마다 빨간점들을 모두 다 먹어야 한다는 것이다. 빨간점들을 모두 먹게 되면 다음 스테이지로 넘어가게 되는데, 미로의 사이즈가 1씩 증가하며 커지게 된다. 그러므로 게임은 시간이 지나면 지날수록 점점 더 어려워진다. 그러면 플레이어 입장에서는 집중력이 흐트러질 수 밖에 없다. 따라서, 게임에서는 집중력에 대한 인센티브를 제공한다. 게임에는 제한시간이 존재한다. 제한시간안에 최대한 많은 스테이지를 클리어하는 것을 목표로 한다. 가장 처음에는 남은 시간 12초부터 시작한다. 남은 시간이 매초 줄어드는 모습을 화면에 출력하여 플레이어의 긴장감을 끌어올렸다. 그리고, 플레이어가 획득하고 있는 현재 스코어는 역동적으로 변화하게 하여 굉장히 많은 점수를 얻고 있다는 생각이 들게 해 의욕을 주려고 했다. 집중력에 대한 인센티브는 다음 스테이지로 넘어갈 때 추가시간의 형태로 주었는데, 추가시간은 현재 미로의 크기의 절반 + 남은시간의 절반만큼 주었다. 미로의 크기에 비례하는 추가시간을 준 이유는 난이도에 따른 기본적인 추가시간이라고 생각하고 주었다. 하지만 여기서 남은시간의 절반을 추가로 부여한 것은 집중력에 대한 인센티브의 관점에서 부여한 것이다. 플레이어가 집중을 해서 스테이지를 금방 깨게 되면 시간이 더 많이 남아있을 것이므로 추가시간을 더 부여받아서 더 게임을 오래 지속할 수 있을 것이다. 하지만, 집중을 하지 않는 플레이어는 스테이지를 클리어 하더라도 상대적으로 적은 추가시간을 부여받아 결국 금방 끝나게 될 것이다. 하지만, 이런 인센티브가 있음에도 불구하고 미로의 크기가 아주 커지게 되면 플레이어입장에서는 더 이상 게임을 하고 싶은 마음이 생기지 않을 수 있다. 이 때를 대비해서 스페이스바를 누르면 힌트를 제공한다. 힌트란 현재 플레이어의 위치에서 남은 빨간점들을 먹을 수 있는 최단 경로를 화면에 보여준 다는 의미이다. 이 힌트는 화면상에 2초정도 그려지며, 서서히 사라지는 애니메이션을 주었다. 원래는 힌트의 개수를 제한하거나 힌트를 얻으면 시간을 차감할 생각이었는데, 이렇게 되면 힌트를 주는 의미가 없어질 것 같아서 힌트는 무한정 얻을 수 있도록 했다. 이렇게 하게 되면, 실력이 좋으면서, 끈기가 있는 플레이어가 아닌 플레이어에 비해 많은 차이로 높은 점수를 얻을 수 있게 되므로, 점수 차이가 촘촘하지 않고 큰 게임을 제작할 수 있게 되었다.
2. <프로젝트 실행 결과>



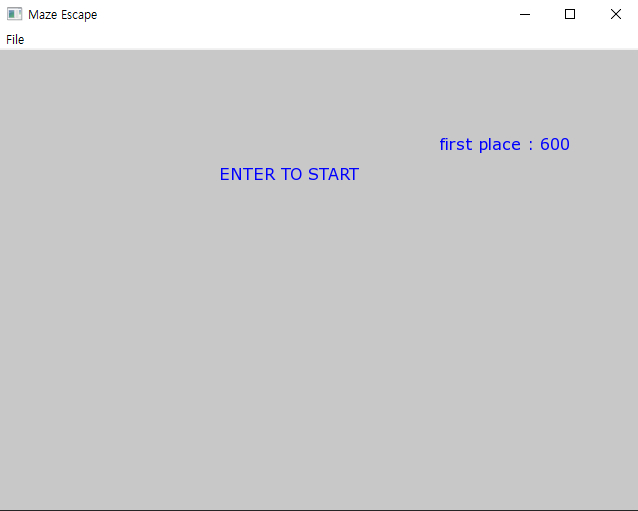
[초기화면]

아무것도 표시되어 있지 않는다. 기록이 하나도 없는 상태이기 때문이다.



[첫번째 게임 중]

첫번째 판이기 때문에 highScore이 없으므로 화면에 출력되지 않는다.



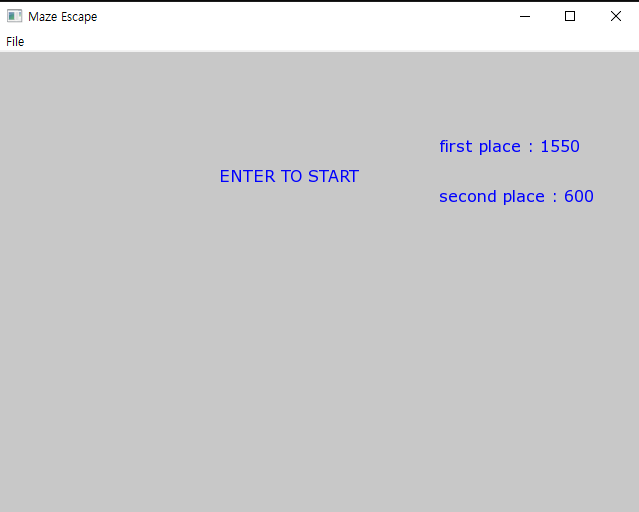
[첫번째 게임 결과]

한판을 했으므로 현재 1등이다.



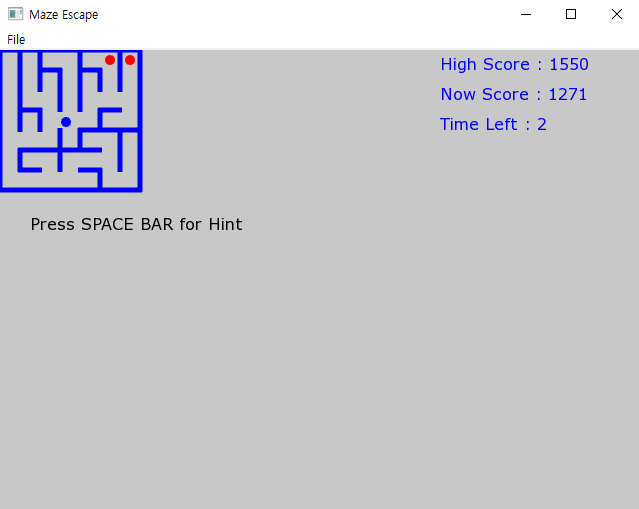
[두번째 게임 중]

힌트를 사용해가면서 게임을 진행한다. 첫번째 판을 진행했기 때문에 highScore 또한 존재한다.



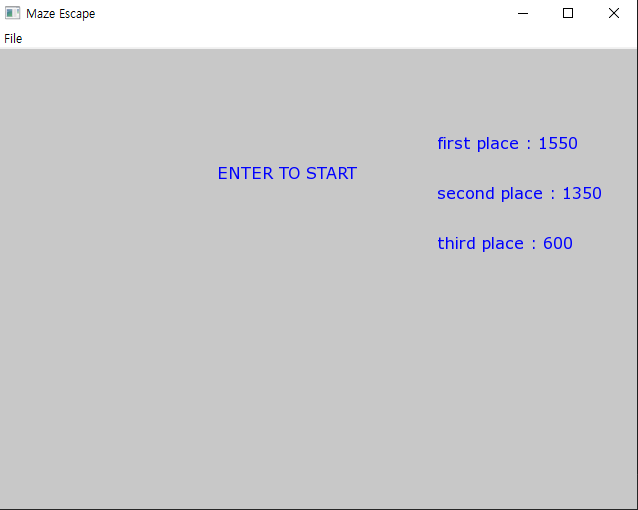
[두번 째 게임 끝난 이후]

두 개의 기록이 내림차순 정렬 되어 있어, 가장 점수 높은 두 점수를 화면에 출력했다.



[세번째 판]

현재 최고기록인 1550점을 깨기는 어려울 것처럼 보인다. 현재 2초가 남은상황에서 1271점이기 때문이다.



[세번째 판 끝난 이후]

예상한 대로 1550점은 넘지 못했지만 1350점을 기록해서 2등자리에서 확인할 수 있다. 원래 600은 2등에서 3등으로 밀려났다.

1. 이번 프로젝트를 진행하면서 c언어의 작동 과정을 더 자세히 알 수 있었고, 파일입출력에 대해서는 완벽히 이해하고 활용할 수 있는 단계까지 올라온 기분이었다. 또한, 오픈프레임워크를 이용하여 눈에 보이는 코딩을 해본적은 처음이었는데, 내가 상상하는 그대로를 컴퓨터화면에서 구현할 수 있다는 사실이 정말 놀라웠다. 기본적으로 컴퓨터공학 설계 및 실험1 과목을 들으며 코드를 짜임새있게 짜는 법을 배운 것 같다. 이 과목을 수강하기 이전에는 스파게티 코드를 자주 짰었고, 디버깅도 하기 어렵게 짰을 뿐 아니라 주석을 다는 습관이 잘 형성되어 있지 않아서, 한번에 원하는 기능을 구현하지 못한 상황에서 디버깅과정이 굉장히 오래걸렸다. 그러나, 주석처리의 중요성을 알게 되었고, 실습시간에 여러 프로젝트를 진행하고, 또 마지막 기말 프로젝트를 혼자의 힘으로 진행하다보니 c언어로 코드를 어떤 식으로 짜는 것이 괜찮은 방법인지 구분이 가는 것 같다. 물론, 아직 숙련되었다고 이야기하기는 어렵지만, 과목을 수강하기 전과 비교한다면 정말 엄청난 성장을 이룬 것 같다. 이번 프로젝트를 진행하면서도, 정말 많은 구글링과, 많은 생각을 한 것 같다. 남이 보기에는 별거 아닌 결과물처럼 보일 지 몰라도 정말 많은 노력을 기울였고, 최선을 다했다고 당당히 이야기할 수 있을 것 같다. 그러나, 이번 프로젝트와 테트리스 프로젝트를 진행하면서 들었던 생각은 이번 프로젝트에서도 테트리스 프로젝트에서 했던 추천시스템과 같은 기능을 구현보고자 생각은 하였으나, 해봤던 과정이기도 하고, 요즘시대에는 테트리스 프로젝트처럼 모든 경우의수를 고려해서 최적의 답을 구하는 것보다 다르게 해답을 구하는 것이 트렌드인 것 같은 느낌을 받았다. 대표적인 예시로는 기계학습이다. 테트리스 프로젝트를 트리형태로 구현하여 모든 경우의수를 고려하여 코드를 짰지만, 기계학습을 이용한다면 작성했던 코드보다 더 적은 코드로, 더좋은 결과물을 낼 수 있지 않을까하는 의문점 및 개선사항이 생겼고, 이에 대해 앞으로 추가로 공부하고 싶어졌다.