

CG final project 10조

최종보고서

openGL을 활용한 SOR 모델러와 3D공간 상의 미로 탐색

과목명	컴퓨터 그래픽스 2분반
팀명	10조 박쥐피플
소속	예술공학부
팀장	20220149 조은비
팀원	20221067 고예경
	20202977 김주환
	20222620 이윤지

목차

1. 기획 및 개발 과정	2
가. 팀원	2
나. 기본 구현 목표	2
다. 팀 구성	2
라. 작업 방식	3
마. 제작 플로우 차트	3
2. 기능 및 실행화면	4
가. 구현 기능 목록 개요	4
나. 실행화면 및 상세 기능목록	5
3. 파일 구성	9
가. 파일 구성	9
나. 사용 라이브러리	9
4. 주요 코드 설명	11
가. 모듈 구성	11
나. 미로 모듈과 카메라 모듈	12
다. 렌더링 모듈 – SOR 모델러	18
라. 렌더링 모듈 – mainApplication	23
마. 렌더링 모듈 – 텍스처 맵핑	25
바. 렌더링 모듈 – Distance 인자 활용	28
사. 렌더링 모듈 – 카메라 오버레이	30
5. 느낀점	31

1. 기획 및 개발 과정

가. 팀원

팀 명	10조 박쥐피플
팀 장	20220149 조은비
팀 원	20221067 고예경
	20202977 김주환
	20222620 이윤지

나. 기본 구현 목표

팀 구성원의 능력과 상황을 고려하여 기본적으로 구현할 기능에 대한 목표를 세웠다.

기본 구현 목표	
■ SOR 모델러	<ul style="list-style-type: none">• 마우스로 점 찍고 회전시키기• 회전 각도 설정• DAT 파일로 저장

■ 미로 맵과 SOR 모델 불러오기
■ 기본적인 렌더 시스템 구성 – glut 라이브러리 내에서 설계
■ 사용자 입력에 따른 카메라 이동
■ 충돌체크

다. 팀 구성

'기본 구현 목표'를 바탕으로 SOR/렌더러 작업 팀과 카메라/맵/충돌 팀으로 각 2인씩 나누어 구성하였다.

SOR/텍스처 팀	고예경, 이윤지
	기본 렌더 시스템 + 텍스처 맵핑 + SOR 모델러
카메라/맵/충돌 팀	김주환, 조은비
	미로맵, SOR 모델 불러오기 + 카메라 이동 + 충돌체크

위의 팀 구성을 바탕으로 작업 전 세부 계획을 세웠다.

- PLAN A) 기본 작업 후, 전 인원이 추가 기능 작업에 합류하여 퀄리티를 높이는데 집중
- PLAN B) 작업이 늦어지는 팀에 합류하여 보충, 추가 기능 작업은 고려

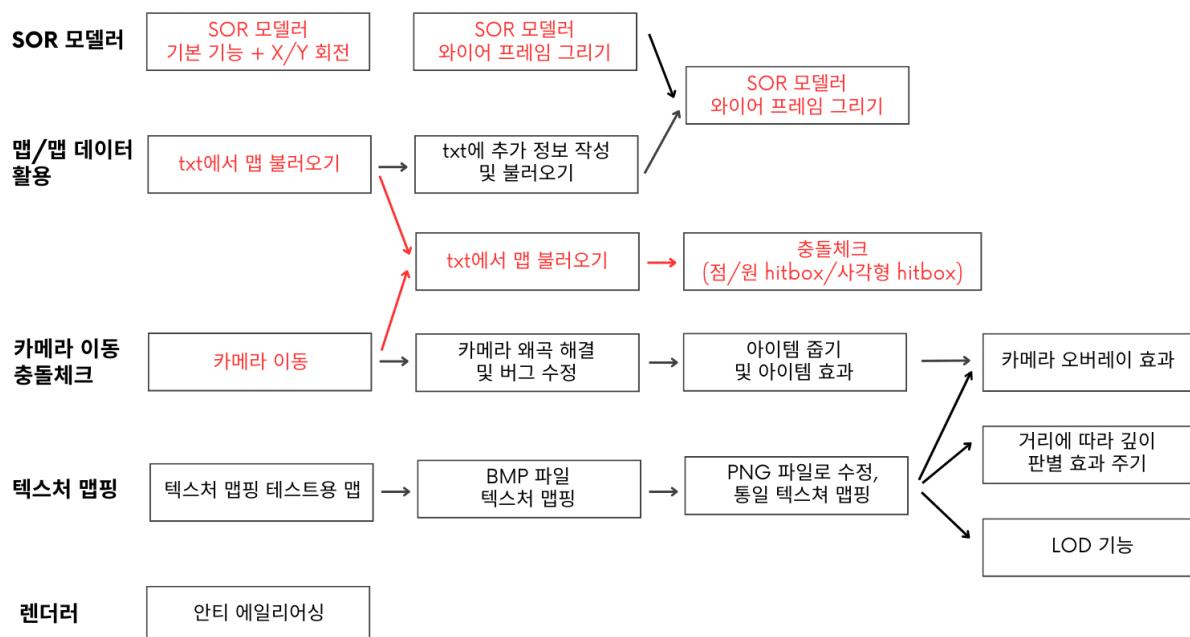
라. 작업 방식

- 노션(Notion): 작업 파일 및 버그/요청 사항 공유
- 팀채팅: 실시간 작업 업데이터, 공유 목표 설정 등 주요 사항 공유

마. 제작 플로우 차트

제작 중간 발표 이후 PLAN B로 작업하였다. 카메라/맵/충돌 팀에서 충돌 작업까지 완료 후 SOR/텍스처 팀에 합류하여 SOR 모델러와 텍스처 맵핑 기능을 완성하였다. 보충작업이 끝난 후, 기존의 PLAN B로 돌아와 만들어진 기능들을 응용하여 웰리티를 높이는데 집중했다.

SOR 모델러, 맵과 맵 데이터 활용 작업, 카메라 이동 및 충돌체크, 그리고 텍스처 맵핑 작업으로 작업 branch를 구분하면 아래와 같다.



2. 기능 및 실행화면

가. 구현 기능 목록 개요

기본 구현 조건 외 추가 기능 목록만 작성하였다.

기능별 항목	구현 기능 세부
SOR 모델러	X/Y 축 기준으로 각도를 지정받아 회전
	와이어 프레임 on/off 토글
	미로 맵 내 여러개의 와이어프레임 SOR 모델 불러오기
	터미널 UI 기능 <ul style="list-style-type: none">◆ Help창, Clear기능, 동작 실행 시 안내 및 예외처리
	여러 개의 모델을 이름 지정해서 저장
카메라로 이동 및 충돌체크	F키로 아이템 상호작용에 의한 아이템 효과 및 카메라 오버레이
	충돌체크 - AABB 충돌
텍스처 맵핑	스카이돔 박스
	벽과 바닥 텍스처링
	거리에 따른 명도 설정
	맵 내에 잔디 심기 및 LOD
	SOR 모델 LOD
렌더러	안티에일리어싱

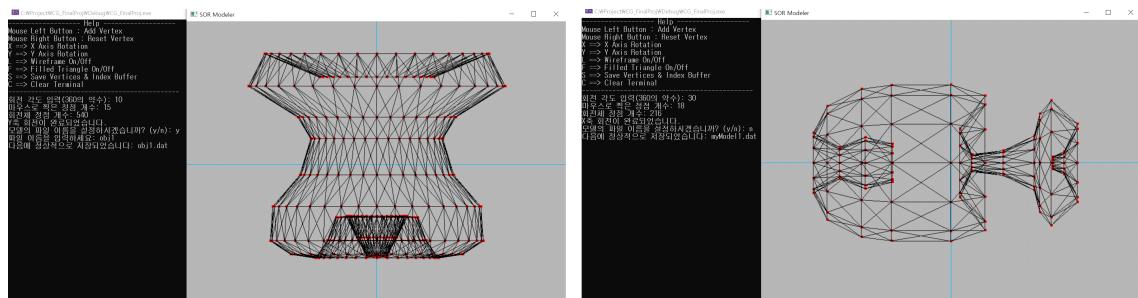
나. 실행화면 및 상세 기능목록

본 구현 기능을 포함한 모든 기능에 대한 항목과 각 항목별 설명 및 실행 화면은 아래와 같다.

1) SOR 모델러

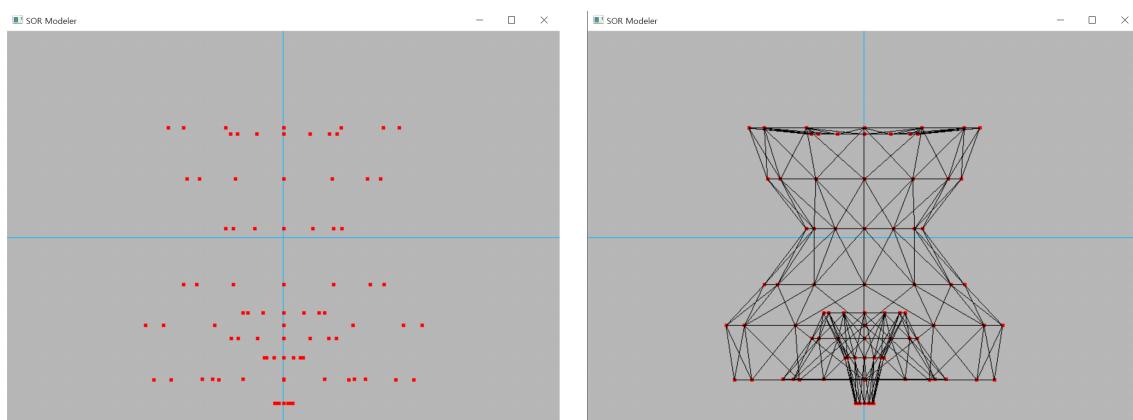
■ X/Y 축 기준으로 각도를 자정받아 회전

SOR 모델러의 기본 기능(정점 입력받고 저장, 화면 출력)을 완성 후 Y축 회전을 추가함



■ 와이어 프레임 그리기

버텍스 버퍼와 인덱스 버퍼를 활용하여 SOR 모델러 내에서 와이어 프레임을 그림



■ 미로 맵 내 여러개의 SOR 모델 불러오기

SOR 모델러에서 만들어진 모델의 DAT 파일과 `drawWireframe()` 함수를 이용하여 미로 맵 내에서 SOR 모델을 import함

2) 맵과 맵 데이터 활용 작업



■ txt파일에 미로의 정보를 담고 import

txt형태의 미로 파일에서 미로 크기, 시작 지점 좌표와 미로 데이터를 담아 OpenGL 뷰포트 내에 불러옴.

■ txt파일에 아이템에 대한 정보 추가하여 import

2~9 사이의 index로 아이템의 위치, 종류에 대한 정보를 담아 OpenGL 뷰포트 내에 불러오고 상호작용 기능에 활용함.

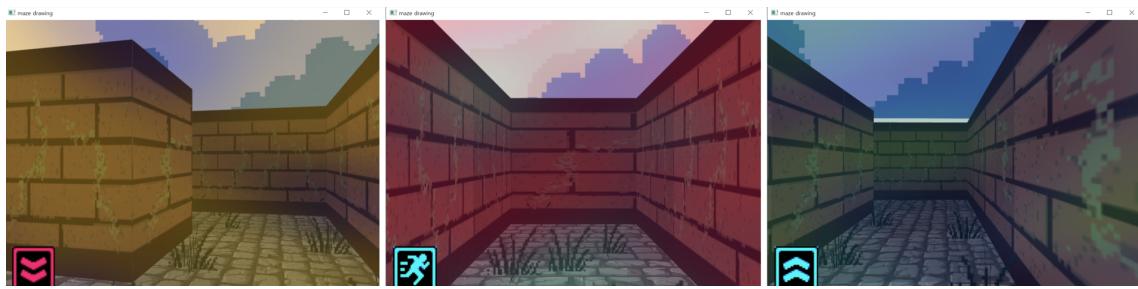
3) 카메라 이동 및 충돌체크

■ WASD 키보드 – 좌우회전, 앞뒤 이동 기능

WASD 시스템 적용, 카메라가 직접 좌우 회전 및 앞뒤 이동하도록 설정함.

■ F키로 아이템 상호작용

SOR모델을 활용한 아이템에 충돌 박스를 설정하여 F키를 통해 아이템 습득이 가능하도록 함. 이때 카메라 오버레이로 효과 아이콘과 화면 이펙트가 나타나도록 설정함.



좌측부터 순서대로 이속 감소 / 벽 통과 / 이속 증가

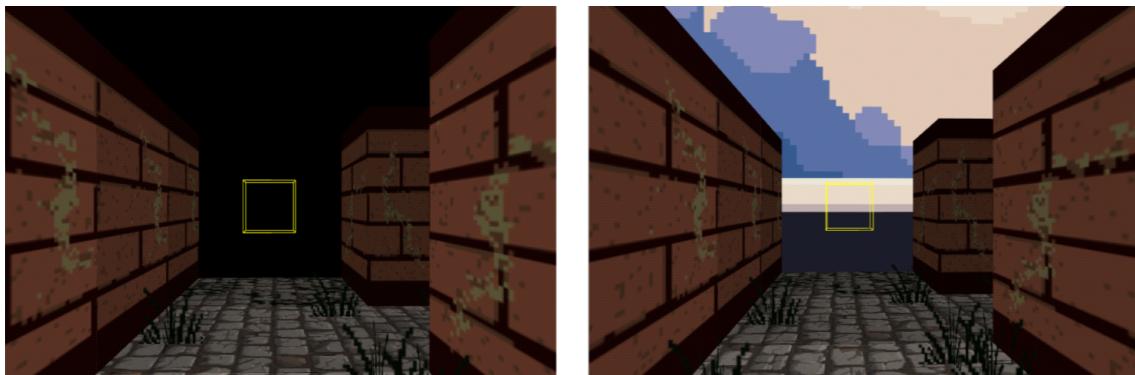
■ 충돌체크 – AABB 충돌

플레이어 주위로 사각형 충돌 박스를 설정, X,Z 좌표의 최대, 최소끼리 비교하여 충돌을 확인하는 기능을 추가함. 벽 충돌 체크 및 아이템 상호작용에 활용함.

4) 텍스처 맵핑

■ 스카이돔 박스

평면 텍스처링을 활용하여 맵 전체에 스카이돔 박스를 추가함.



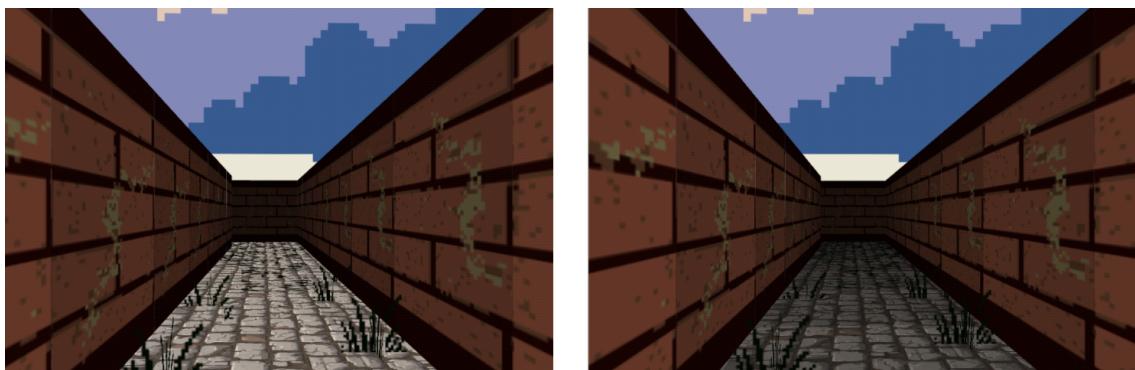
■ 벽 텍스처링, 거리에 따른 명도 설정

텍스처 로더 클래스를 설계하여 벽에 png 텍스처를 맵핑함. 플레이어(카메라)의 위치와 블록의 거리를 계산하여 거리에 따라 벽 큐브의 색이 어두워지도록 하여 깊이감을 부여함.



■ 바닥 텍스처링, 거리에 따른 명도 설정

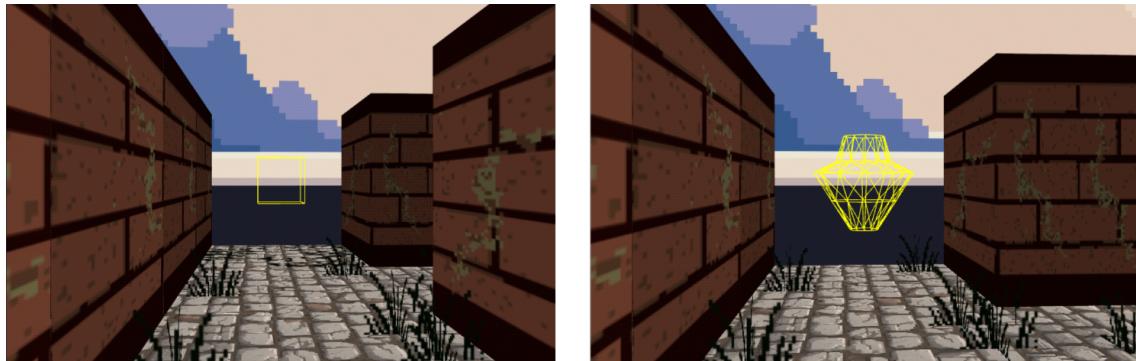
텍스처 로더 클래스와 텍스처 맵핑된 평면 클래스를 설계하여 바닥에 png 텍스처를 맵핑함. 플레이어(카메라)의 위치와 블록의 거리를 계산하여 거리에 따라 바닥 평면의 색이 어두워지도록 하여 깊이감을 부여함.



■ 맵 내에 잔디 심기 및 LOD

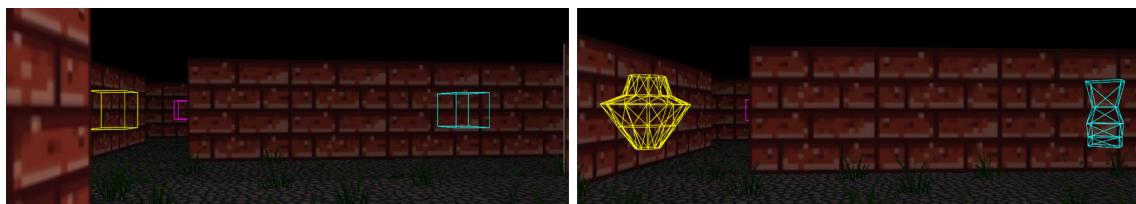
텍스처 로더 클래스와 텍스처 맵핑된 평면 클래스를 설계하여 바닥에 잔디를 배치함.

플레이어(카메라)의 위치와 블록의 거리를 계산하여 거리에 따라 잔디가 단순한 형태로 렌더링 되도록 LOD 기능을 부여함.



■ SOR 모델 LOD

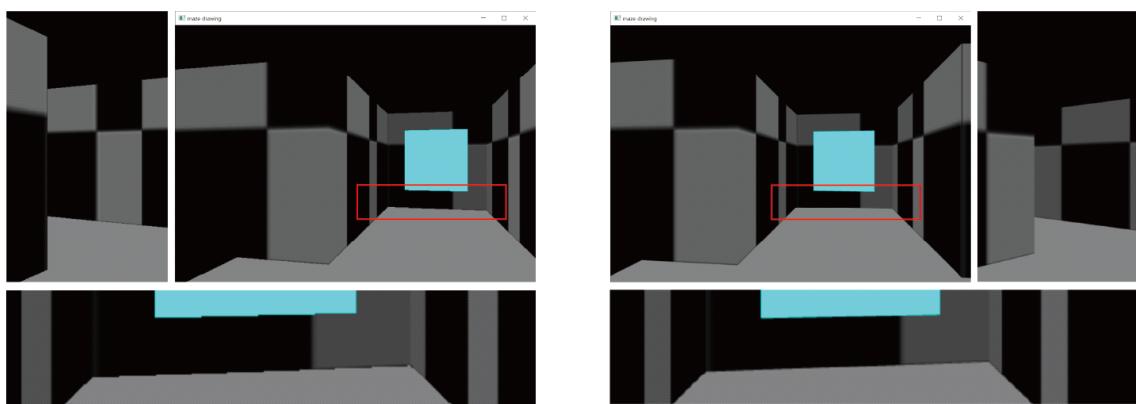
텍스처 버퍼와 인덱스 버퍼 정보를 활용하여 SOR 모델을 미로에 불러옴. 플레이어(카메라)의 위치와 블록의 거리를 계산하여 거리에 따라 SOR모델이 단순한 형태로 렌더링 되도록 LOD 기능을 부여함.



5) 렌더러

■ 안티에일리어싱

더블 버퍼를 활용하여 안티에일리어싱 기능을 추가함.



각 기능에 대한 코드와 알고리즘에 대한 설명은 ‘주요 코드 설명’ 항목을 참조.

3. 파일 구성

가. 파일 구성

FinalProject.zip 파일의 구성은 아래와 같다.

상위 파일명	하위 파일명	내용
/Bin	/resource	미로 길, 벽, 스카이박스, 아이템 효과 오버레이를 구성하는데 필요한 png파일, 미로 정보를 불러오는 txt파일을 포함함.
	/SORobj	SOR모델러에서 export한 모델의 dat파일을 포함함.
	Maze.exe	미로 게임 실행 파일
	SOR_Modeler.exe	SOR 모델러 실행 파일
/Source	main.h	미로 프로그램에 대한 헤더파일
	main.cpp	미로 프로그램에 대한 cpp파일
	SOR.cpp	SOR 모델을 만들고 저장하는 프로그램이다.

나. 사용 라이브러리

미로 게임 프로그램과 SOR모델러에 사용된 라이브러리의 목록은 아래와 같다.

1) main.cpp 사용 라이브러리

라이브러리	내용
<iostream>	표준 입출력을 위한 라이브러리로, 동작 실행 안내 및 디버깅을 위해 사용함.
<fstream>	파일 입출력 라이브러리로, txt파일에서 정보를 받아올 때 사용함.
<vector>	데이터의 동적 할당을 위한 라이브러리로, 미로의 주요 데이터를 가변적으로 다루기 위해 사용하였다.
<string>	문자열을 동적으로 다루는 라이브러리로, 문자열을 용이하게 다루기 위해 사용함.
<cmath>	수학 연산에 필요한 라이브러리로, 미로에서 카메라의 회전 등에 사용함.
<ctime>	시간에 관한 라이브러리로, 미로에서 아이템 효과의 타이머 기능을 위해 사용함.
<GL/glut.h>	CG Rendering Pipeline의 전체적인 기능을 활용함.
<GL/stb_image.h>***	png파일을 이용한 Texture mapping에 활용함.

*** https://github.com/nothings/stb/blob/master/stb_image.h 에서 stb_image.h를 다운받은 후, 프로젝트에 링킹된 freeglut 라이브러리가 포함된 폴더의 include/GL 폴더에 넣어 설치한다.

2) SOR.cpp 사용 라이브러리

라이브러리	내 용
⟨iostream⟩	표준 입출력을 위한 라이브러리로, 동작 실행 안내 및 디버깅을 위해 사용함.
⟨fstream⟩	파일 입출력 라이브러리로, SOR모델러에서 dat파일을 만들 때 사용함.
⟨sstream⟩	문자열을 다루는 라이브러리로, SOR모델러에서 여러 모델에 대한 문자열을 입력받아 저장하기 위해 사용함.
⟨vector⟩	문자열을 동적으로 다루는 라이브러리로, 문자열을 용이하게 다루기 위해 사용함.
⟨string⟩	문자열을 동적으로 다루는 라이브러리로, 문자열을 용이하게 다루기 위해 사용함.
⟨cmath⟩	수학 연산에 필요한 라이브러리로, SOR 회전 등에 사용함.
⟨GL/glut.h⟩	CG Rendering Pipeline의 전체적인 기능을 활용함.

4. 주요 코드 설명

가. 모듈 구성

미로 게임을 보다 효율적으로 구성하기 위해 기능과 역할을 중심으로 미로 모듈, 카메라 모듈, 렌더링 모듈로 주요 모듈을 나누어 설계했다. 반복적으로 사용되거나 기능이 유사한 내용은 함수나 클래스로 모듈화하여 중복을 최소화하였다.

각각의 기능과 객체들을 나누고, 하나의 모듈로 묶어내는 작업에 대한 고민을 지속적으로 공유하였는데, 이렇게 모듈화된 설계로 여러 기능들을 용이하게 추가할 수 있는 유연성과 확장성을 보장할 수 있었다. 또한 함수와 클래스를 다양한 기능에 효율적으로 활용, 코드의 재사용성을 극대화하였다.

아래는 미로 게임과 SOR 모델러에 사용된 모듈에 대한 목록이다.

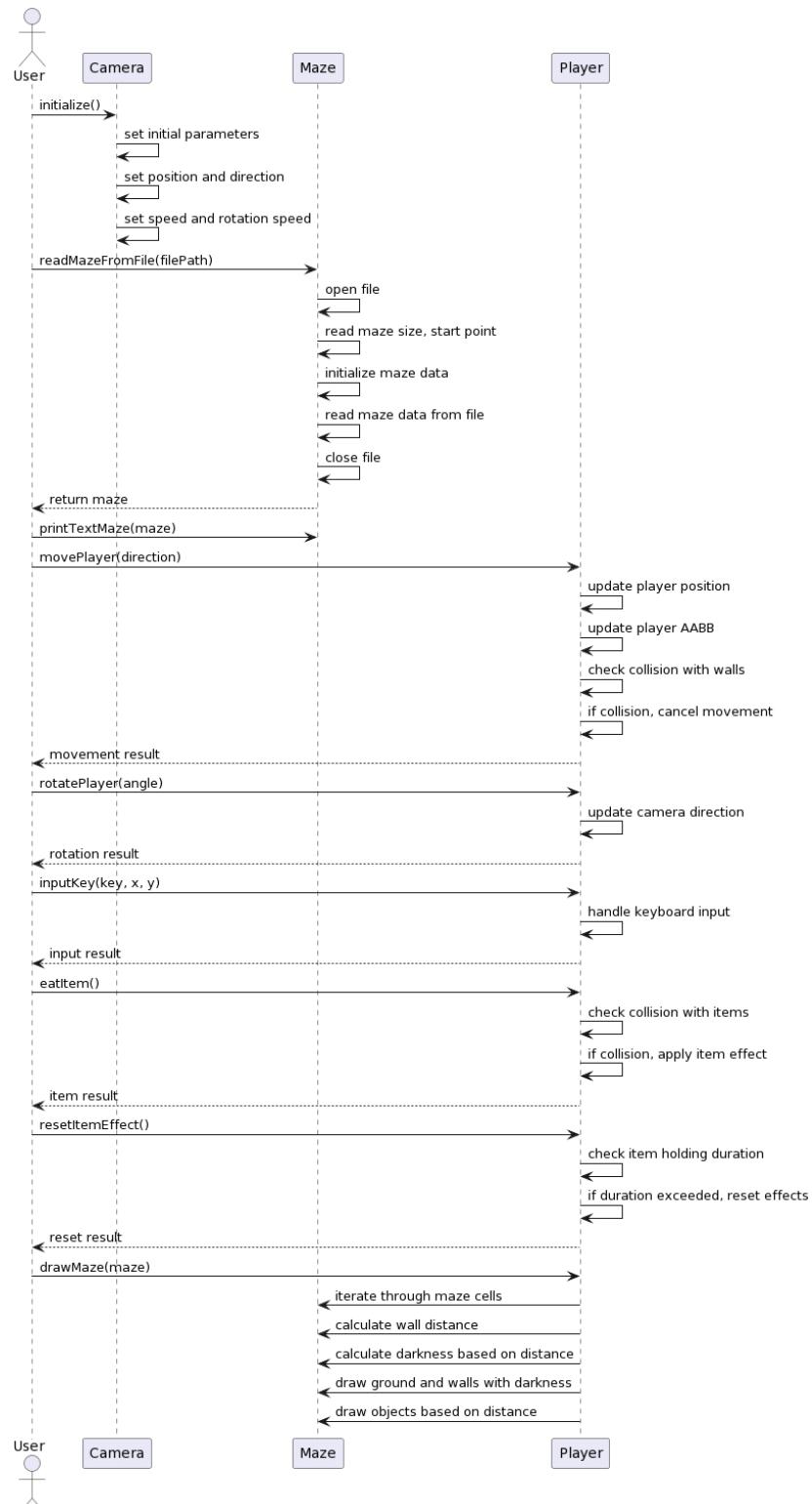
모듈	내용
미로 모듈	미로 데이터 읽기
	미로 데이터를 바탕으로 벽과 아이템 draw
카메라 모듈	키보드 콜백으로 이동
	충돌체크
렌더링 모듈	SOR 모델러, SOR 모델 클래스
	텍스처 맵핑
	Distance 인자 활용
	카메라 오버레이

나. 미로 모듈과 카메라 모듈

1) 미로 게임의 시퀀스 다이어그램

미로 게임은 카메라를 통해 보이는 화면과 플레이어의 시야를 일치시킨 FPS(1인칭)뷰를 제공한다.

아래의 다이어그램은 유저가 미로 게임 프로그램을 시작하고 조작하여 상호작용하는 일련의 과정을 그린 것이다.



(1) 카메라 초기화 `initialize()`

유저가 프로그램을 시작하면 카메라의 매개변수, 위치, 방향, 속도 및 회전 속도를 초기화시킨다.

(2) 미로 읽기 `readMazeFromFile(filePath)`

txt형태의 미로 파일에서 미로 크기, 시작 지점 좌표와 미로 데이터를 읽어온다.

(3) 키보드 입력 처리 `inputKey(key, x, y)`

유저의 입력(WASD)을 처리하여 `movePlayer(direction)`과 `rotatePlayer(angle)`를 호출한다.

(4) 플레이어 이동 `movePlayer(direction)`, 플레이어 회전 `rotatePlayer(angle)`

유저 입력(WASD키)을 통해 플레이어를 이동 · 회전시킨다. 플레이어의 위치와 AABB 충돌박스를 업데이트하여 벽과의 충돌 여부를 확인한다.

(5) 아이템 습득 및 효과 적용 `eatItem()`, `applyItemEffect()`

플레이어와 아이템의 충돌을 확인하고, `applyItemEffect()` 를 호출하여 아이템들의 효과를 각각 적용시킨다.

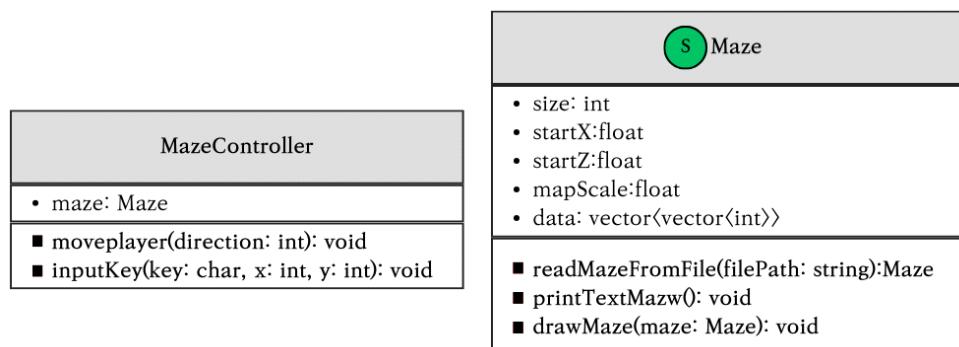
(6) 미로 그리기 `drawMaze(maze)`

플레이어가 현재 위치한 미로를 draw한다. 플레이어의 좌표값과 미로의 셀까지의 거리 `wallDistance`을 계산하고, 이에 따라 다시 명암을 계산하여 미로를 그린다.

2) 미로 모듈 – maze module

■ 활용 기능: txt파일에 미로의 정보를 담고 import / txt파일에 아이템에 대한 정보 추가하여 import

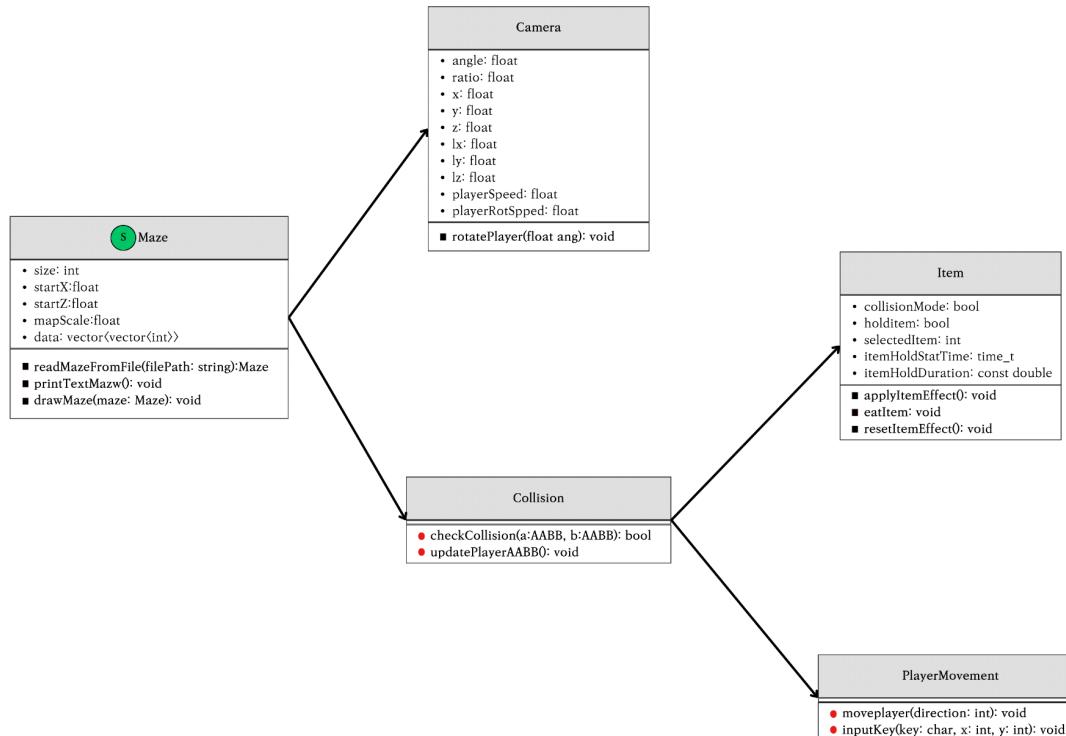
위의 다이어그램은 Maze 구조체와 Maze를 인자로 받는 함수와 Maze maze; 로 선언된 구조체를 활용하는 함수들을 나타내었다.



미로 모듈의 UML

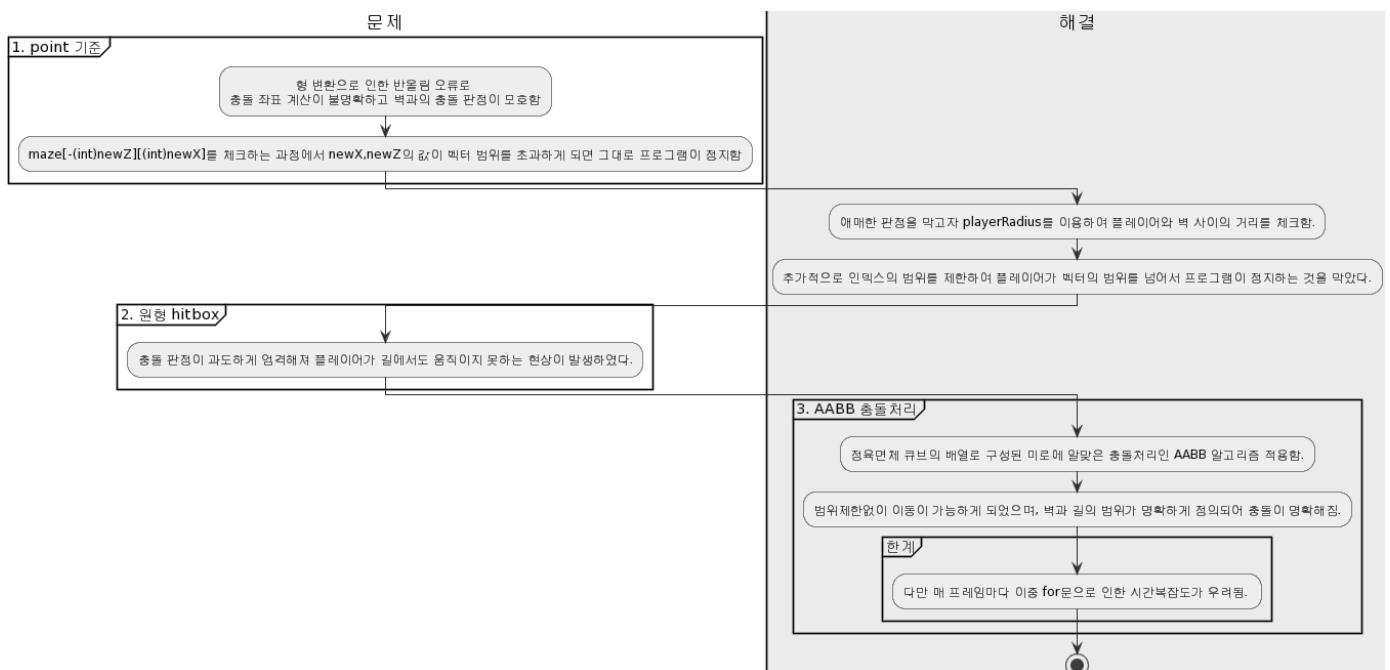
3) 카메라 모듈 – camera module / Collision check

- 활용 기능: 키보드 콜백으로 이동 / 충돌체크



카메라 모듈 내의 플레이어 이동과 충돌체크 UML

카메라 모듈은 유저로부터 키보드 입력을 받아 충돌을 체크하고, 플레이어의 좌표를 이동시키는 동작을 담당한다. 이때 사용된 충돌 체크 알고리즘은 여러 시행착오를 거쳐 point 기준, 원형 hitbox, 그리고 사각 hitbox를 모두 구현한 후 가장 충돌 판정이 자연스러운 AABB 충돌을 활용한 사각 hitbox를 최종적으로 사용하였다.



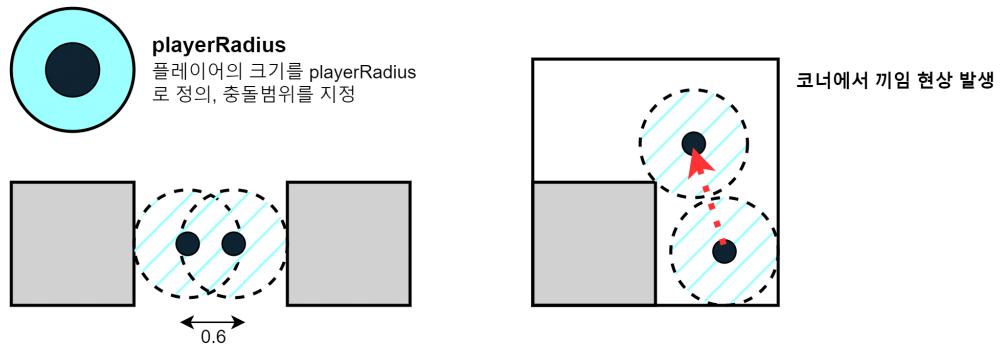
(1) point 기준

- 플레이어의 크기를 지정하지 않고, 좌표값으로 받아 위치를 판별한다. 2차원 maze 벡터의 data에서 벽과 길의 정보를 받아와 벽에서는 플레이어가 움직이지 않도록 위치를 갱신하지 않는다.

■ 문제

- ◆ 플레이어와 좌표사이의 거리를 계산하므로 판정에 오차가 발생
- ◆ float로 계산된 값을 int형 인덱스와 맵핑시키는 과정에서 반올림된 값 만큼의 오차가 발생
- ◆ `maze[-(int)newZ][(int)newX]`에서 `newX, newZ`의 값이 벡터 범위를 초과할 경우 인덱스 범위 초과 에러 발생

(2) 원형 hitBox



■ 해결

point에 플레이어의 크기 `playerRadius` 만큼의 반경으로 플레이어에 크기를 주었다. 1번에서와 마찬가지로 벽은 2차원 maze 벡터의 data를 활용하였다. 추가적으로 읽어오는 벡터 인덱스의 범위를 제한하여 인덱스 초과 에러로 프로그램이 정지하는 것을 방지했다.

■ 문제

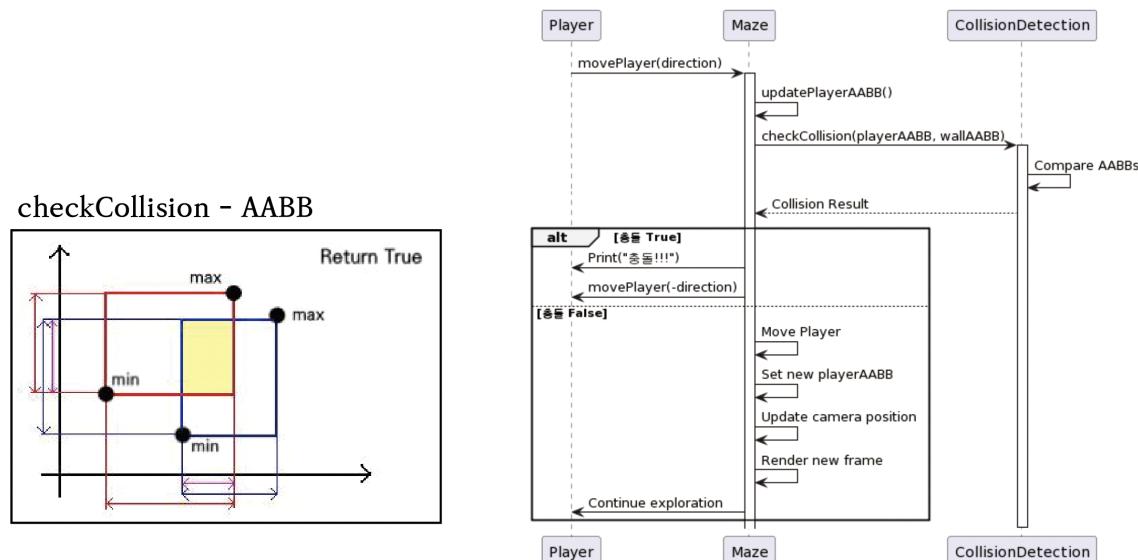
- ◆ float로 계산된 값을 int형 인덱스와 매핑시키는 과정에서 반올림된 값 만큼의 오차가 (이하 벡터 맵핑 오차)발생
- ◆ 충돌판정이 이전보다 엄격해졌으며, 사각형의 벽과 원형의 플레이어 hitbox의 차이로 인해 복잡한 길목에서 플레이어가 움직이지 못하는 현상(이하 낑김 현상) 발생

(3) AABB 충돌처리

■ 해결 - AABB(Axis Aligned Bounding Box) 충돌

플레이어에 직육면체의 hitbox를 AABB구조체를 활용하여 지정하였다.

AABB구조체에 직사각형의 (x,z)좌표 최소값, 최대값 두개를 저장, 두 AABB 구조체의 x, z최대 최소가 둘다 겹치면 true 값을 리턴하여 충돌을 판정한다.



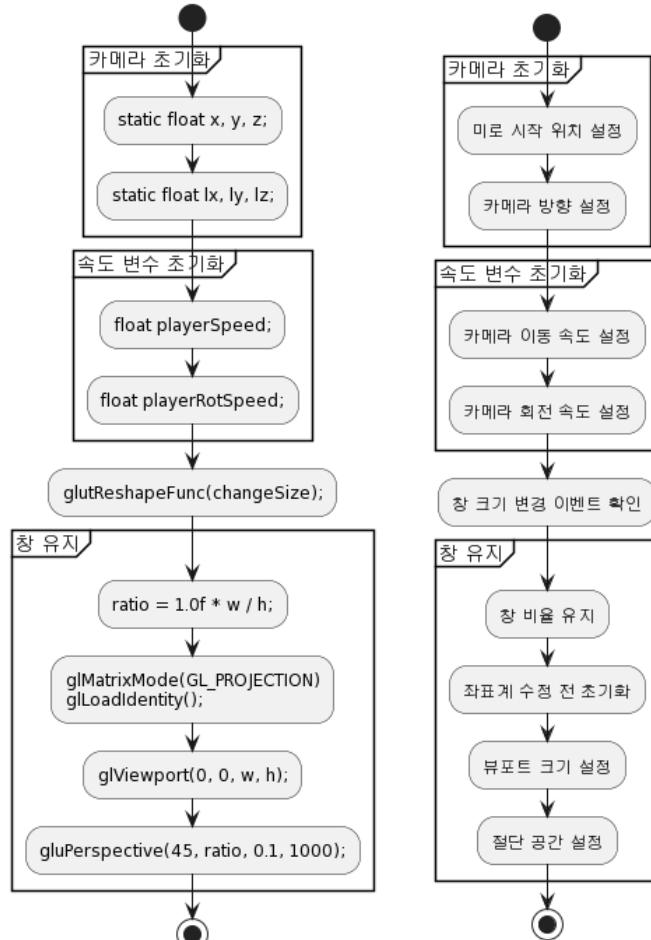
[좌측]AABB 충돌체크 / [우측]카메라 모듈 내의 플레이어 이동 및 충돌체크 시퀀스 다이어그램

유저가 플레이어를 이동시키기 위해 키보드 키를 입력하면 키보드 콜백에 의해 `checkCollision()`이 호출된다. 이때 `updatePlayerAABB()`를 통해 이동 후 플레이어의 AABB를 갱신하고, for문을 돌며 모든 벽에 대해 충돌 boolean을 확인한다. 이때 리턴된 값이 True면 충돌로 판정, 이동 전의 AABB 값으로 되돌리고, False면 갱신된 AABB 값을 유지한다.

AABB 충돌 알고리즘을 적용한 결과, maze 벡터의 인덱스를 활용하지 않아 (1)과 (2)번의 충돌알고리즘에서 발생한 벡터 맵핑 오차 문제가 해결되었다. 또한, 벽과 플레이어의 충돌 범위가 명확하게 정의되어 충돌 오차를 줄였으며, 낑김 현상을 해결할 수 있었다.

4) 카메라 초기화

- 활용 기능: 키보드 콜백으로 이동 / 충돌체크



카메라 초기화 시퀀스 다이어그램

(1) 카메라 초기화

카메라의 회전각 `angle`, 카메라의 시작좌표 `(x, y, z)`와 카메라가 바라보는 방향 `(lx, ly, lz)`를 초기화한다.

(2) 속도 변수 초기화

플레이어의 이동속도 `playerSpeed`와 플레이어의 회전속도 `playerRotSpeed`를 초기화 한다.

플레이어가 아이템을 습득하면 아이템의 효과에 따라 두 변수의 값을 조절한다.

(3) 창 유지

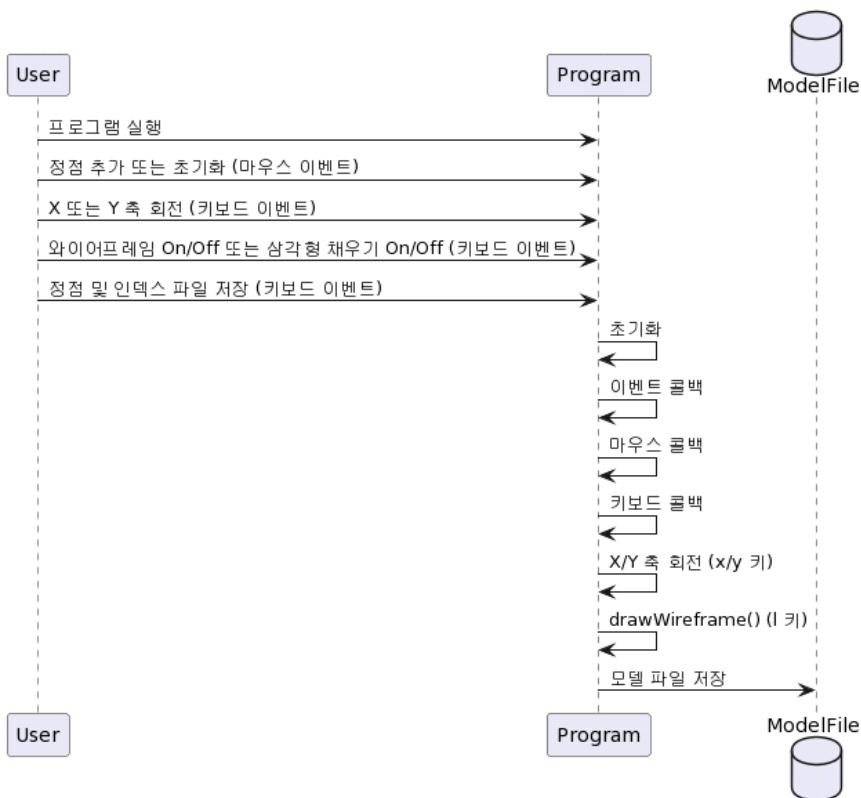
카메라 모듈에서 최종 렌더링되는 뷰포트 윈도우의 창 비율과 뷰 모드를 초기화한다. 카메라 오버레이가 적용되는 스코프 외에는 해당 설정을 기본값으로 가진다.

다. 렌더링 모듈 – SOR 모델러

1) SOR 모델러 기본 기능

- 활용 기능: SOR모델러 기본기능 / X/Y 축 기준으로 각도를 지정받아 회전

SOR 모델러는 정점을 입력하고 X또는 Y축 회전으로 만들어진 회전체를 모델로 저장하는 프로그램이다. 사용자는 마우스 이벤트를 통해 정점을 입력하고 키보드 이벤트(x또는y)를 통해 X/Y축 회전을 시킬 수 있다. 이후 와이어프레임을 키보드 입력을 통해 On/Off 할 수 있으며 정점 및 인덱스버퍼를 ModelFile로 저장하게 된다.



SOR 모델러의 시퀀스 다이어그램

(1) 모델 저장 `SaveModel()`

사용자에게 파일 이름을 입력 받거나 자동으로 생성하여 모델을 .dat 파일로 저장하는 함수. `mPoint` 벡터에 저장된 정점들과 `indexBuffer` 벡터에 저장된 인덱스들을 dat파일에 저장한다.

(2) 인덱스 버퍼 생성 `createIndexBuffer()`

`indexBuffer`를 새로 생성하는 함수. 현재 정점의 수(`originalVertices`)와 회전 후 정점의 수(`rotatedVertices`)를 기반으로 적절한 인덱스를 생성하여 `indexBuffer`에 저장한다.

(3) 인덱스 버퍼 출력 `printIndexBuffer()`: `indexBuffer`에 저장된 내용을 콘솔에 출력하는 함수.

(4) 정점 그리기 `drawPoints()`: `mPoint` 벡터에 저장된 정점들을 점으로 그리는 함수.

(5) 정점 초기화 `resetPoints()`: `mPoint` 벡터와 `indexBuffer`를 초기화하여 정점을 리셋하는 함수.

(6) 정점 축 회전 `rotatePointsXAxis()`, `rotatePointsYAxis()`

정점들을 X 또는 Y 축을 중심으로 주어진 각도만큼 회전시키는 함수.

(7) 와이어프레임 그리기 `drawWireframe()`

와이어프레임 표시 여부에 따라 `mPoint`와 `indexBuffer`를 이용하여 선분을 그리는 함수.

(8) 삼각형 채우기 `drawFilledTriangles()`

와이어프레임과 함께 사용자가 선택한 경우, 반투명 회색으로 정점들을 이용하여 삼각형을 그리는 함수.

(9) 모델러 x/y축 표시 `drawAxes()`: X 및 Y 축을 그리는 함수.

(10) 화면 업데이트 `display()`: OpenGL 창의 화면을 업데이트하고 다시 그리는 함수.

(11) 마우스 이벤트 콜백 `mouseCallback()`

마우스 클릭 이벤트에 대한 콜백 함수로, 왼쪽 버튼 클릭 시 정점을 추가하고, 오른쪽 버튼 클릭 시 모든 정점을 리셋하는 역할을 함.

(12) 키보드 이벤트 콜백 `keyboardCallback()`

키보드 이벤트에 대한 콜백 함수로, 사용자의 입력에 따라 X 또는 Y 축 회전, 와이어프레임 및 반투명 회색으로의 표면 채우기 등을 수행하는 함수.

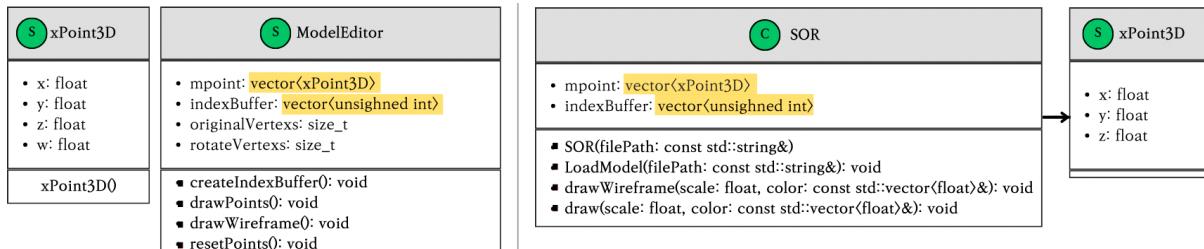
(13) 터미널 도움말 출력 `myHelp()`: 사용자에게 키보드 및 마우스 이벤트에 대한 도움말을 출력하는 함수.

(14) `main()`

프로그램의 진입점 및 초기화를 담당하는 함수. 윈도우 생성, 초기화 및 이벤트 핸들링에 관련된 함수들을 호출함.

2) 버텍스 버퍼, 인덱스 버퍼

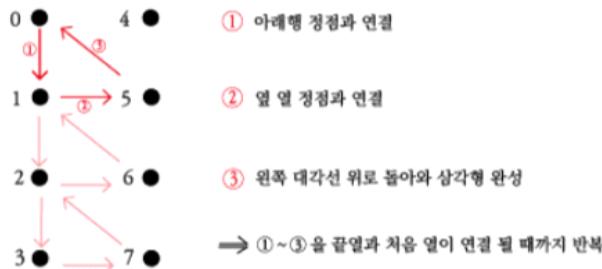
- 활용 기능: 와이어 프레임 그리기 / 미로 맵 내 여러개의 SOR 모델 불러오기



[좌측] SOR 모델러의 SOR 모델과 관련된 자료구조 / [우측] 미로 게임의 SOR 클래스 구조

위의 다이어그램은 SOR모델에 사용된 자료구조를, 우측의 다이어그램은 미로 게임 내에서 사용된 SOR 자료구조를 나타낸 것이다. SOR 모델러를 제작하고 SOR 모델러에서 만든 회전체를 미로로 불러오는 과정에서는 버텍스 버퍼와 인덱스 버퍼의 개념을 적극적으로 활용했다.

■ 인덱스 버퍼 생성 알고리즘



좌측의 그림은 인덱스 버퍼의 생성 알고리즘을 간단히 그림으로 나타낸 것이다. 회전체의 정점을 열과 행으로 구분하여 아래행과 연결, 우측 열과 연결, 좌측 대각선 위로 돌아와 삼각형 완성을 반복하여 삼각형으로 와이어프레임을 차례로 완성하도록 하였다.

```

for (size_t i = 0; i < r; ++i) {
    for (size_t j = 0; j < n; ++j) {
        // 아래로 연결 (제일 마지막 열의 경우 예외 처리)
        if (j != n - 1) {
            indexBuffer.push_back(i * n + j);
            indexBuffer.push_back(i * n + (j + 1) % n);
        }

        // 옆으로 연결
        indexBuffer.push_back(i * n + j);
        indexBuffer.push_back((i + 1) % r * n + j);
        // 대각선 연결 (한 방향으로만)

        if (j < n - 1) {
            indexBuffer.push_back(i * n + j);
            indexBuffer.push_back((i + 1) % r * n + (j + 1));
        }
    }
}

```

〈저장된 DAT 파일 구조〉

DAT 파일	obj.dat
• 베텍스 버퍼	mpoint: vector<xPoint 3D>
• 인덱스 버퍼	indexBuffer: vector<unsigned int>

EX)



정점의 (x,y,z) 위치 좌표를 저장한 xPoint구조체를 순서대로 베텍스 버퍼 mpoint(vector<xPoint 3D>)에 저장했다.

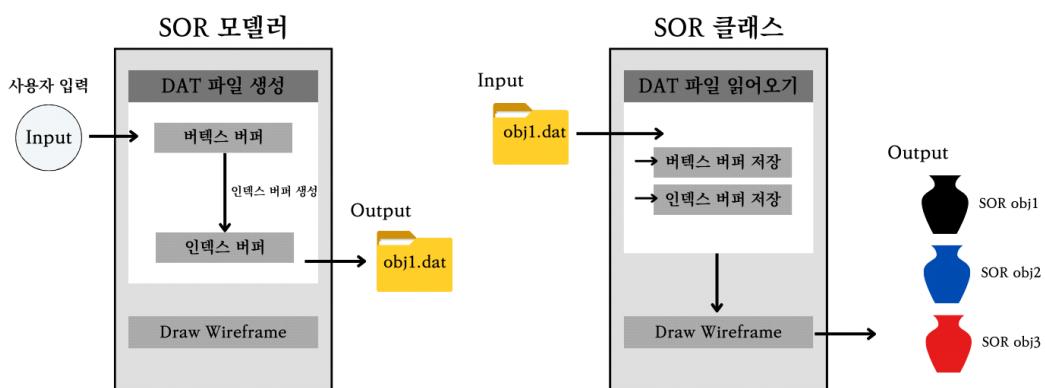
정점을 잇는 순서를 인덱스로 저장한 인덱스 버퍼를 indexBuffer(vector<unsigned int>)에 저장했다.

이렇게 생성된 인덱스 버퍼를 베텍스 버퍼의 내용과 함께 dat 파일로 저장하여 SOR 모델러에서 만들어진 SOR 모델을 추출하였다.

3) SOR 모델 클래스

- 활용 기능: 미로 맵 내 여러개의 SOR 모델 불러오기

SOR 모델러에서 작성된 베텍스 버퍼 mpoint와 인덱스 버퍼 indexBuffer를 .dat 파일로 저장하여 미로 게임 내에서 resource로 사용했다. 이때 베텍스 버퍼에는 회전체 정점의 위치가, 인덱스 버퍼에는 정점을 잇는 순서를 담은 정보가 있으므로 LoadModel()와 drawWireframe()기능만을 SOR 모델을 불러오는 클래스의 메소드로 정의하였다.



[좌측] SOR 모델러의 SOR 모델 I/O 시스템 개요 [우측] 미로 게임의 SOR 클래스 I/O 시스템 개요

SOR 모델 클래스의 LoadModel()로 dat파일을 읽어와 각각 벡터 자료구조로 저장하고, drawWireframe()으로 indexBuffer에 따라 와이어프레임의 형태로 draw할 수 있다.

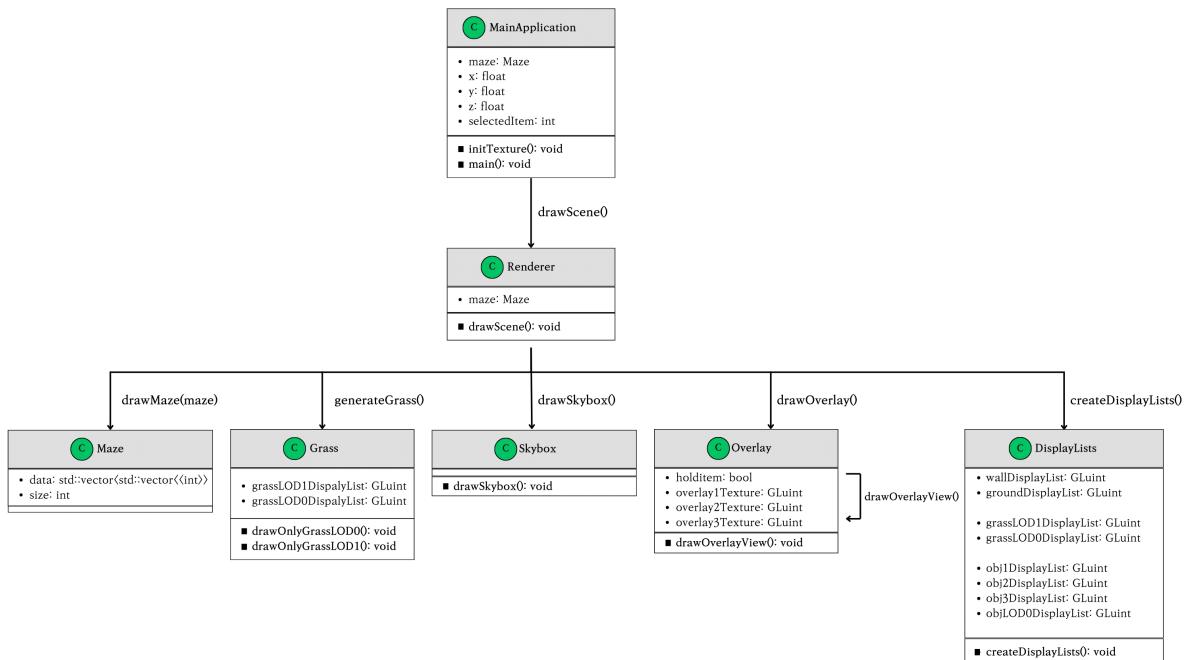
미로 코드 상에서 SOR 클래스의 각 개체는 SOR obj1(filepath)의 형태로 선언하고, obj1.draw()로 뷰포트 상에 호출하여 활용할 수 있도록 설계했다.

클래스 SOR	
멤버 변수	<ul style="list-style-type: none"> ■ xPoint3D 구조체: x, y, z, w 로 이루어진 3D좌표를 저장한다. ■ std::vector<xPoint3D> mpoint 회전체 모델의 전체 정점의 좌표를 저장한 버텍스 버퍼를 벡터로 저장한다. ■ std::vector<unsigned int> indexBuffer 회전체 모델의 와이어프레임을 그리는 정점 인덱스의 순서쌍을 저장한 인덱스 버퍼를 벡터로 저장한다.
메소드	<ul style="list-style-type: none"> ■ 생성자 SOR(const std::string& filePath) 회전체 모델의 전체 정점의 좌표를 저장한 버텍스 버퍼를 벡터로 저장한다. ■ LoadModel(const std::string& filePath) 파일 경로에서 모델 데이터를 읽어와 mpoint와 indexBuffer에 저장한다. ■ drawWireframe(float scale, const std::vector<float>& color) 모델을 와이어프레임으로 렌더링한다. 각 정점의 좌표에 스케일을 적용하고 지정된 컬러로 라인을 그린다. (*추가 계획 중 <code>drawTriangleFill()</code> 추가를 위해 <code>draw()</code>와 분리해둠) ■ draw(float scale, const std::vector<float>& color): <code>drawWireframe</code> 메소드를 호출하여 와이어 프레임을 그린다.

■ 사용 예시

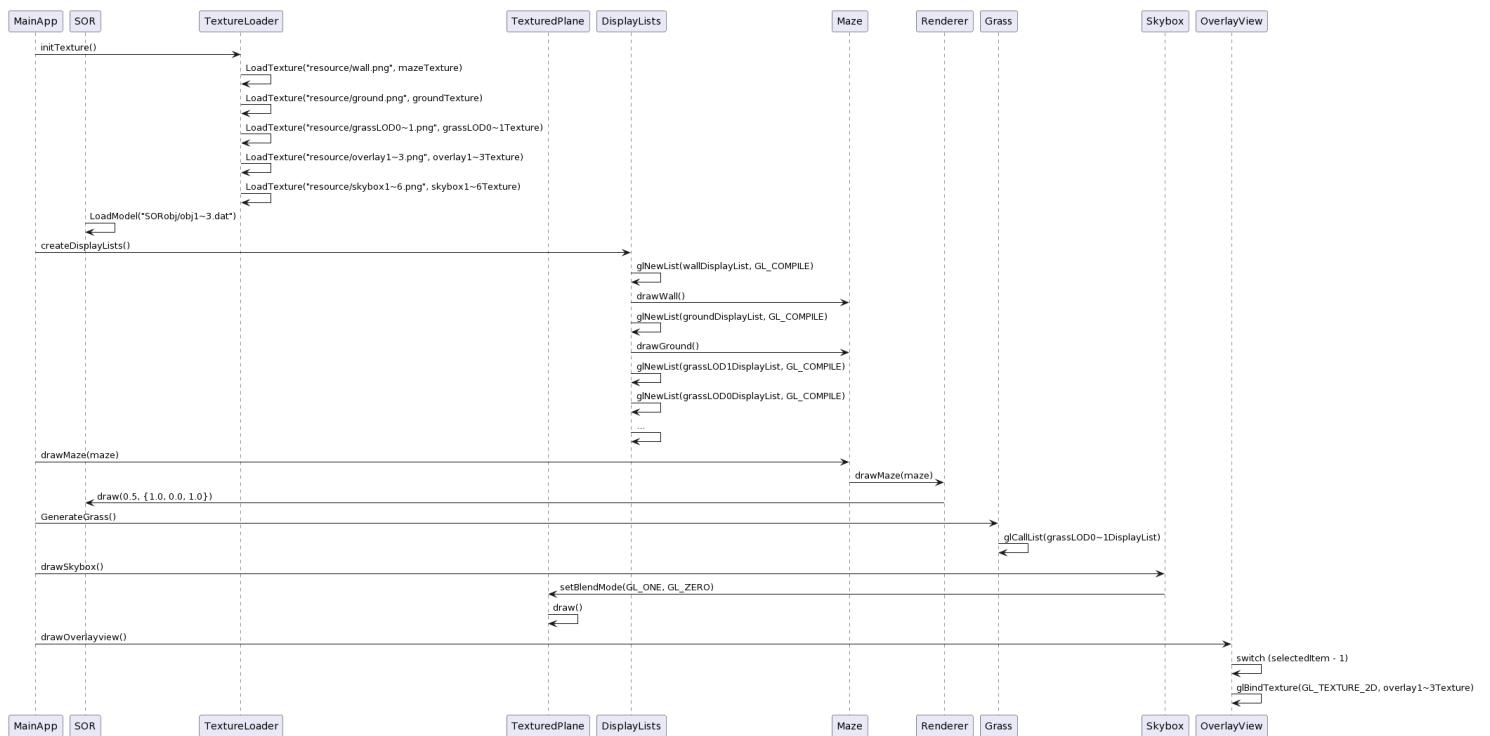
```
obj1DisplayList = glGenLists(1);
glNewList(obj1DisplayList, GL_COMPILE);
SOR obj1("SORobj/obj1.dat");
obj1.draw(0.5, { 1.0, 0.0, 1.0 });
glEndList();
```

라. 렌더링 모듈 – mainApplication



렌더링 모듈 구성 UML

1) 렌더링 모듈의 시퀀스 다이어그램



렌더링 모듈의 시퀀스 다이어그램

(1) MainApp

`TextureLoader`로 여러 텍스처를 로드한다. `SOR`로 `LoadModel` 메서드를 호출하여 SOR 모델러로 만든 dat파일을 모델로 로드한다. `DisplayLists`로 wall, ground, grass, object의 디스플레이 리스트를 선언한다.

(2) 미로 그리기 `drawMaze(maze)`

`Maze` 구조체에서 정보를 받아와 각 셀마다 벽과 길에 대한 정보와 아이템에 대한 정보를 받아와 draw한다. 이때 플레이어의 좌표값과 미로의 셀까지의 거리 `wallDistance`를 계산하여 벽과 바닥의 명암과 오브젝트의 LOD(Level of Detail)를 결정한다.

(3) 잔디 생성 `GenerateGrass()`

`Maze` 구조체에서 미로의 크기에 대한 정보를 받아와 전체 바닥면 위에 잔디를 draw한다. 이때 플레이어의 좌표값과 미로의 셀까지의 거리 `wallDistance`를 계산하여 잔디의 LOD(Level of Detail)를 결정한다.

(4) 스카이 박스 draw

`TexturedPlane`을 활용하여 각 면마다 스카이박스 텍스처를 맵핑, 정육면체를 디스플레이 리스트에 저장 후 호출하여 스카이 박스를 그린다.

(5) 카메라 오버레이 뷰 draw

선택된 아이템에 따라 텍스처를 변경하여 오소그래픽 뷰로 카메라 오버레이 효과를 draw한다.

2) 디스플레이 리스트 목록

벽, 바닥, 잔디 등 화면에 반복적으로 렌더링되는 모든 개체들은 디스플레이 리스트로 선언, 저장하여 `glCallList(DisplayList)`로 호출하여 draw 하였다.

■ 미로 벽과 바닥 디스플레이 리스트

```
GLuint wallDisplayList;  
GLuint groundDisplayList;
```

■ 잔디 LOD별 디스플레이 리스트

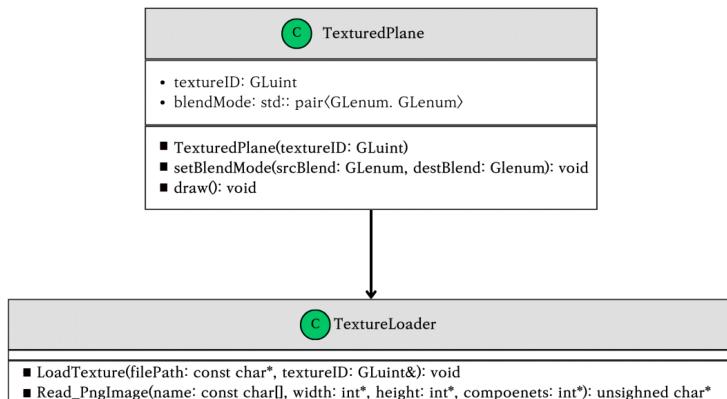
```
GLuint grassLOD0DisplayList;  
GLuint grassLOD1DisplayList;
```

■ SOR 오브젝트(아이템)의 LOD별 디스플레이 리스트

```
GLuint obj1DisplayList;  
GLuint obj2DisplayList;
```

```
GLuint obj3DisplayList;  
GLuint objLOD0DisplayList;
```

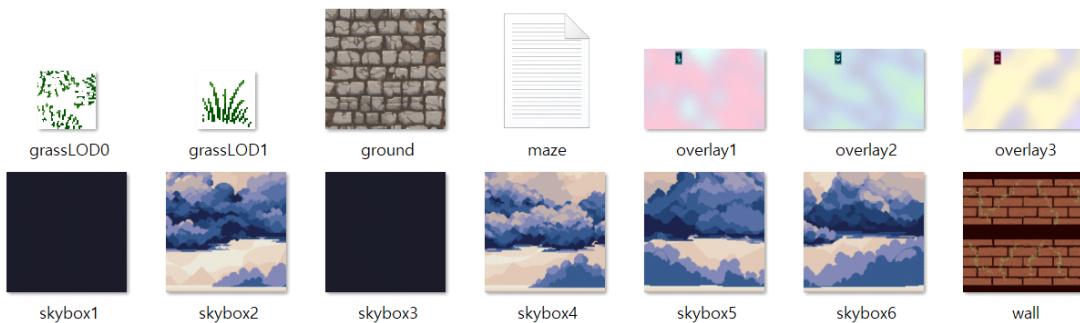
마. 렌더링 모듈 - 텍스처 맵핑



텍스처 맵핑을 담당하는 클래스의 다이어그램

TextureLoader와 TexturedPlane를 활용하여 OpenGL을 사용하여 텍스처를 로딩하고 텍스처를 맵핑한 평면을 draw하였다.

1) 텍스처 리소스 목록



■ 미로 벽과 바닥 텍스처

```
GLuint mazeTexture;  
GLuint groundTexture;
```

■ 잔디 LOD별 텍스처

```
GLuint grassLOD0Texture;  
GLuint grassLOD1Texture;
```

■ 아이템별 카메라 오버레이 텍스처

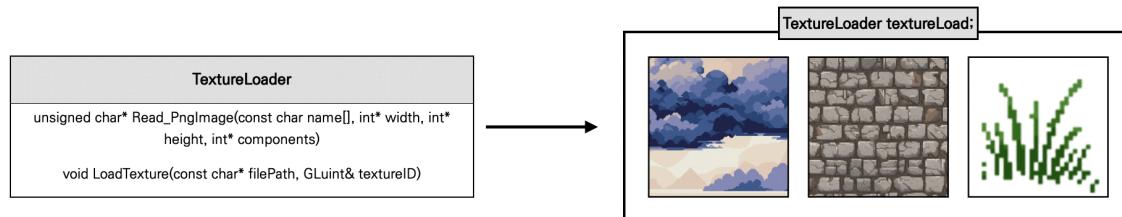
```
GLuint overlay1Texture;  
GLuint overlay2Texture;  
GLuint overlay3Texture;
```

■ 스카이박스 텍스처

```
GLuint skybox1Texture;  
GLuint skybox2Texture;  
GLuint skybox3Texture;  
GLuint skybox4Texture;  
GLuint skybox5Texture;  
GLuint skybox6Texture;
```

2) TextureLoader

■ 활용 기능: png 텍스처 import 및 init



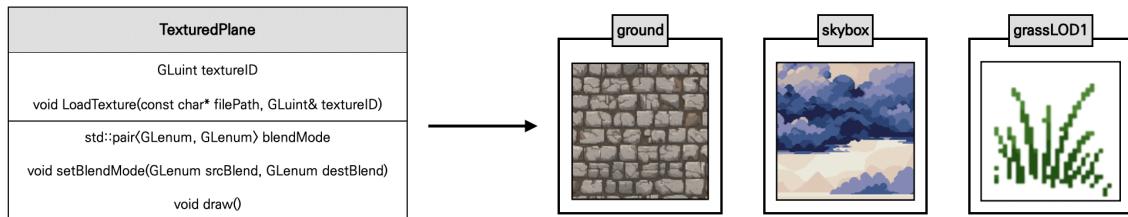
클래스 SOR	
멤버 변수	<ul style="list-style-type: none"><code>unsigned char* Read_PngImage(const char name[], int* width, int* height, int* components)</code> PNG 이미지를 읽어오는 함수. STB 라이브러리를 사용한다. PNG 파일을 읽어오면 이미지의 너비(width), 높이(height), 구성 요소(components)를 리턴한다.
메소드	<ul style="list-style-type: none"><code>void LoadTexture(const char* filePath, GLuint& textureID)</code> 주어진 파일 경로에서 PNG 이미지를 읽어와 OpenGL 텍스처로 로딩한다. 텍스처의 ID를 리턴한다.

■ 사용 예시

```
TextureLoader textureLoad;  
void initTexture() {  
    textureLoad.LoadTexture("resource/wall.png", mazeTexture);  
    textureLoad.LoadTexture("resource/ground.png", groundTexture);  
}
```

3) TexturedPlane

- 활용 기능: 스카이박스 생성, 바닥 블록 생성, 잔디 블록 생성



클래스 SOR	
멤버 변수	<ul style="list-style-type: none">■ <code>GLuint textureID</code>: 텍스처의 OpenGL ID■ <code>std::pair<GLenum, GLenum> blendMode</code> 해당 텍스처와 평면의 블렌딩 모드를 인자로 전달한다.
메소드	<ul style="list-style-type: none">■ <code>TexturedPlane(GLuint textureID)</code>: 주어진 텍스처 ID로 객체를 초기화한다.■ <code>void setBlendMode(GLenum srcBlend, GLenum destBlend)</code> 블렌딩 모드를 설정한다.■ <code>void draw()</code>: 텍스처가 맵핑된 평면을 그리는 함수. 블렌딩 모드와 텍스처를 활성화한 후 평면을 그린다.

- 사용 예시

```
void drawOnlyGrassLOD0() {
    glPushMatrix();
    TexturedPlane grassLOD0(grassLOD0Texture);
    grassLOD0.setBlendMode(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    grassLOD0.draw();
    glPopMatrix();
}
```

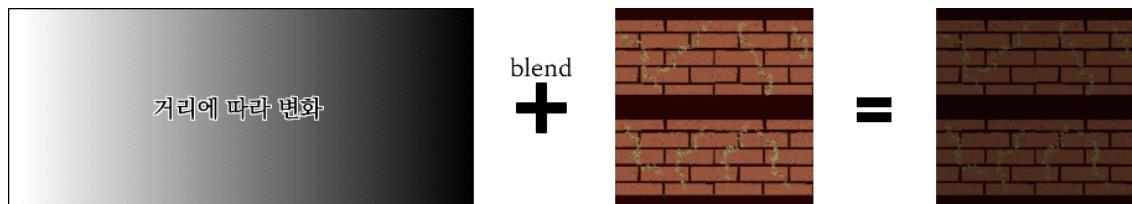
바. 렌더링 모듈 - Distance 인자 활용

glut 라이브러리를 활용하여 유저가 미로 속에서 벽의 공간감을 느낄 수 있도록 만들기 위해 고민하였다. 조명과 빛 처리를 포함하는 복잡한 기법 대신, 플레이어를 기준으로 거리에 따라 명도차이에 변화를 주어 깊이감을 부여하는 간단하면서도 효과적인 방법을 고안했다.

Distance 인자를 활용하여 깊이감을 주는 방법은 레이캐스팅 알고리즘에 아이디어를 착안, 보다 단순화하여 구현하였다. 그리고 이를 한 번 더 응용하여 먼 거리에 있는 모델은 단순화하여 표현하는 LOD 기능을 추가하였다. 이를 통해 시야에서 먼 객체들은 단순화하여 표현하여 렌더링 성능을 향상시키면서도, 전체적인 공간의 깊이와 입체감을 부여하였다.

1) 깊이감 부여

- 활용 기능: 벽 블록 거리에 따른 명도 설정 / 바닥 블록 거리에 따른 명도 설정



(1) wallDistance 계산

거리(wallDistance)는 미로의 셀의 좌표(j, i) 와 플레이어의 좌표(x, z) 사이의 직선거리로 계산한다.

```
wallDistance = sqrt((x - j) * (x - j) + (z + i) * (z + i));
```

(2) darkness 계산

wallDistance에 가중치를 주고, 상수 1.0f를 더한 값을 역수로하여 명도값을 계산한다.

```
float darkness = 1.0f / (wallDistance * 0.5f + 1.0f);
```

(3) 블록에 darkness를 blend하여 적용

벽과 바닥 블록의 BlendMode를 (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`)로 설정하여 큐브 혹은 평면의 색과 텍스처의 색이 blend되도록 하였다. 이때 `darkness`를 `glColor3f()`의 인자로 받아 큐브의 색이 거리가 멀어질수록 어둡게 표현되도록 설정하였다.

2) LOD 기능

- 활용 기능: 잔디의 LOD 기능/SOR 모델 LOD 기능

(1) wallDistance 계산

거리(wallDistance)는 미로의 셀의 좌표(j, i) 와 플레이어의 좌표(x, z) 사이의 직선거리로 계산한다.

```
wallDistance = sqrt((x - j) * (x - j) + (z + i) * (z + i));
```

(2) LOD판별

if문에 의해 wallDistance에 따라 적절한 LODdisplaylist를 호출한다.

```
if (wallDistance >=0.0 && wallDistance <=3.0)
else if (wallDistance >3.0 && wallDistance <=8.0)
```

잔디와 SOR 모델 오브젝트(아이템)의 LOD 기능을 강조하기 위해 상당히 가까운 거리인 3칸/8칸을 기준으로 LOD를 달리 설정하였다.

■ 잔디 LOD

화면	거리	내용
 LOD1 grass	LOD1: 0.0 < 거리 ≤ 3.0	90도 각으로 겹쳐진 두 개의 평면으로 구성된 잔디 묶음이 바닥 블록의 위에 렌더링된다.
 LOD0 grass	LOD0: 3.0 < 거리 ≤ 8.0	단일 평면이 바닥 블록의 위에 렌더링 된다.
	8.0 < 거리	잔디를 렌더링하지 않는다.

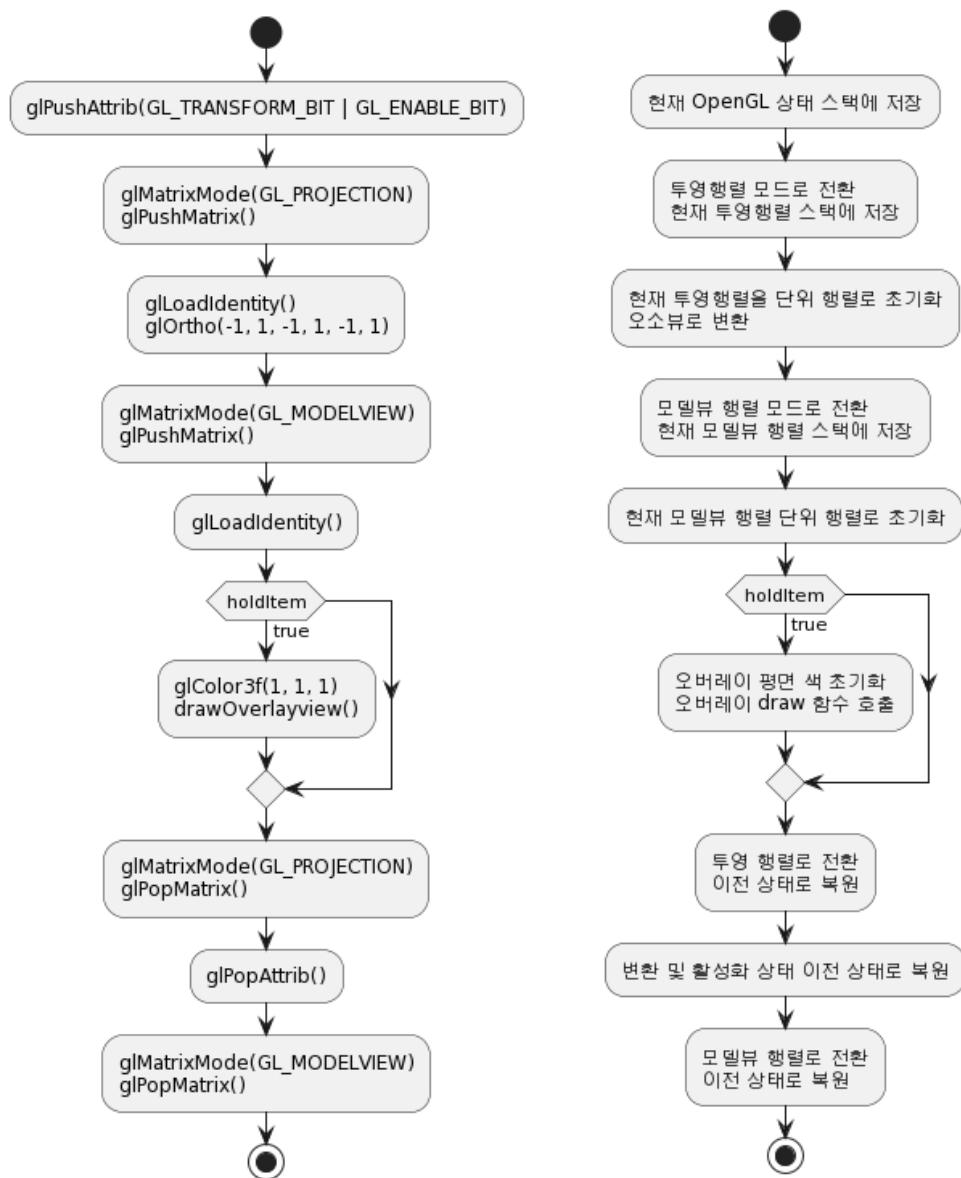
■ SOR LOD

화면	거리	내용
 LOD1 obj	LOD1: 0.0 < 거리 ≤ 3.0	SOR 클래스로 로드된 원본 회전체가 렌더링 된다.
 LOD0 obj	LOD0: 3.0 < 거리 ≤ 8.0	원본 회전체와 동일한 색상의 정육면체 와이어프레임 큐브가 렌더링 된다.
	8.0 < 거리	SOR 모델을 렌더링하지 않는다.

사. 렌더링 모듈 - 카메라 오버레이

카메라 오버레이는 2D 요소를 3D 장면 위에 그리는 방식을 사용하였다. 카메라 오버레이를 제외한 미로 렌더링 모듈은 3D 장면으로 투영 행렬과 모델뷰 행렬 설정이 다르기 때문에 각각의 행렬들을 스택에 저장한 후 2D 요소를 그린 뒤 복원시키는 과정을 거쳤다.

아래는 카메라 오버레이 draw 함수의 플로우 차트이다.



카메라 오버레이의 작동 시퀀스 다이어그램

5. 느낀점

openGL glut 라이브러리로 환경을 제한하였기 때문에 단순한 방법으로 시각적으로 차별성을 만들 수 있도록 많은 고민과정을 거쳤다. 만들어낸 기능은 최대한 많은 부분에서 재사용하려 했는데, 특히 텍스처를 입힌 평면은 바닥, 잔디, 스카이박스 등 다양한 목적으로 곳곳에 배치하여 활용하였다.

충돌 처리를 유저 입장에서 이질감 없이 자연스럽게 만들기 위해 여러 시도를 해야했다. 그 외에도 인덱스 버퍼 개념에 대한 이해를 바탕으로 인덱스 버퍼를 생성하는 알고리즘을 완전히 새로 만들어야 했었는데, 완성도를 올리기 위해 많은 시행착오를 거쳐야 했다.

초기 계획 수립 단계에서 기본 구현 기능과 추가 구현 기능을 구분하여 목표를 세웠기 때문에, 해당 목표에 따라 카메라, 미로 맵, 텍스처, 렌더링으로 크게 모듈화하여 작업할 수 있었다. 따라서 모듈별로 작업과 수정을 진행할 수 있어 공동작업이 원활하게 이루어졌으며, 진행도가 낮은 부분을 빠르게 보충하는 등 문제상황에 유연하게 대처할 수 있었다.

다만 추가 기능 작업 단계에서 기능이 확장될 수 있음을 예상하지 못해, 미리 코드의 유연성을 확보하지 못했고, 그때그때 필요에 따라 클래스나 기타 구조체로 랩핑하여 사용하는 방식으로 진행했다. 이러한 문제로 마무리 단계에서 중복선언과 헤더파일 중복 포함 에러와 전역 변수의 관리가 어려워져 최종 파일에서는 하나의 main 파일로 구성하게 되었다.

미로 게임 프로젝트에 대해 설명하기 위해 코드를 직접 제시하는 대신 표와 다이어그램을 활용하여 알고리즘 및 기능의 구현 과정을 간결하게 설명하고자 했다. 또한, 다수의 팀원이 참여한 프로젝트인 만큼 현재 진행 상황과 발생한 문제 상황을 명확하게 전달하기 위해 그림이나 다이어그램을 적극적으로 활용하였다. 이러한 소통 방법은 상황에 대한 이해도를 높이는데 효과적이었으며, 문제를 시각화하기 위해 정리하고 도식화하는 과정 중에 해결점을 찾기도 했다.

